VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH KOLEKTIVNÍCH KOMUNIKACÍ AKCELEROVANÝ POMOCÍ GPU

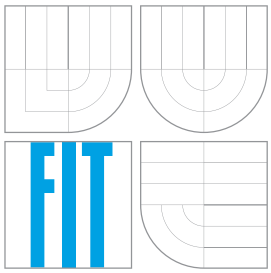
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK TYRALA

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH KOLEKTIVNÍCH KOMUNIKACÍ AKCELEROVANÝ POMOCÍ GPU

EVOLUTIONARY DESIGN OF COLLECTIVE COMMUNICATIONS ACCELERATED BY GPUS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK TYRALA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2012

Abstrakt

Tato práce provádí analýzu existující aplikace implementující evoluční algoritmus pro plánování kolektivních komunikací a navrhuje možnosti její akcelerace s využitím obecných výpočtů na grafických čipech (GPU). V práci je obsažen teoretický úvod do problematiky systémů na čipu, plánování kolektivních komunikací a podrobnější popis evolučních algoritmů. Práce dále zkoumá architektury GPU a paměťovou hierarchii grafických karet z pohledu OpenCL. Na základě analýzy zaměřené na časovou náročnost jednotlivých částí aplikace je proveden návrh paralelního zpracování hodnotící funkce fitness a odhad dosažitelného zrychlení. Stěžejní část práce popisuje implementaci navrženého řešení se zaměřením na využité optimalizace. Práce přináší srovnání původního řešení na CPU a paralelního provedení na GPU. V práci je popsána implementace distribuce výpočtu mezi různá zařízení podporovaná standardem OpenCL a jsou diskutovány výhody, omezení a další možnosti akcelerace výpočtu na základě jeho distribuce na heterogenních výpočetních systémech.

Abstract

This thesis provides an analysis of the application for evolutionary scheduling of collective communications. It proposes possible ways to accelerate the application using general purpose computing on graphics processing units (GPU). This work offers a theoretical overview of systems on a chip, collective communications scheduling and more detailed description of evolutionary algorithms. Further, the work provides a description of the GPU architecture and its memory hierarchy using the OpenCL memory model. Based on the profiling, the work defines a concept for parallel execution of the fitness function. Furthermore, an estimation of the possible level of acceleration is presented. The process of implementation is described with a closer insight into the optimization process. Another important point consists in comparison of the original CPU–based solution and the massively parallel GPU version. As the final point, the thesis proposes distribution of the computation among different devices supported by OpenCL standard. In the conclusion are discussed further advantages, constraints and possibilities of acceleration using distribution on heterogenous computing systems.

Klíčová slova

evoluční návrh, genetický algoritmus, kolektivní komunikace, systém na čipu, propojovací síť, architektura GPU, obecné výpočty na GPU, OpenCL, paralelizace, akcelerace, optimalizace

Keywords

evolutionary design, genetic algorithm, collective communications, system on a chip. interconnection network, GPU architecture, general-purpose computing on GPU, OpenCL, parallel computation, acceleration, optimization

Citace

Radek Tyrala: Evoluční návrh kolektivních komunikací akcelerovaný pomocí GPU, diplomová práce, Brno, FIT VUT v Brně, 2012

Evoluční návrh kolektivních komunikací akcelerovaný pomocí GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiří Jaroš, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Tyrala
22. května 2012

Poděkování

Děkuji vedoucímu práce panu Ing. Jířímu Jarošovi, Ph.D. za projevenou ochotu, poskytnutí odborné pomoci a užitečných rad během tvorby této práce. Dále děkuji Vítězslavu Tesařovi za pomoc při tvorbě grafických schémat a ilustrací.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Paralelní zpracování se při řešení složitých výpočtů v oblasti výpočetních systémů začalo uplatňovat již kolem roku 1970 v komerčních systémech společností IBM a Honeywell [26]. Současný vývoj vysoce výkonných výpočetních systémů se výrazně zaměřuje na tzv. systémy na čipu (SoC). Jedná se typicky o vestavěné systémy vyvinuté pro konkrétní aplikaci. Tyto vysoce paralelní systémy mohou sestávat z desítek až stovek výpočetních jednotek. Takto masivní paralelizace s sebou přináší problém komunikace mezi jednotlivými výpočetními jednotkami. Komunikace se může snadno stát uzkým místem celého systému a značně snižovat efektivitu výpočtu. Dopad tohoto problému lze zmírnit zvolením vhodné propojovací sítě mezi vypočetními jednotkami a přesným časovým plánem provedení všech komunikací.

Vytvoření přesného časového plánu, tedy zajištění alokace linek pro jednotlivé zprávy v čase, není u složitých topologiích triviální úkol. Podle [19] se jedná o NP-úplný problém. Neexistuje tedy obecný algoritmus, který by pro libovolnou topologii vytvořil optimální plán komunikace v polynomiálním čase. Pro nalezení optimálního nebo alespoň téměř optimálního řešení se proto často využívají metody tzv. *soft computingu* [1].

Evoluční algoritmy jsou často využívaným nástrojem při řešení optimalizačních úloh. Jedná se zejména o hledání globálního maxima, resp. minima dané funkce. V případě kolektivních komunikací se pak jedná o nalezení globálního minima funkce, vyjadřující počet kolizí při alokaci zdrojů. Do sdílených zdrojů lze zařadit například komunikační kanály nebo buffery.

Tato práce navazuje na evoluční návrh plánování kolektivních komunikací popsaný v [11] a navrhuje tento postup akcelerovat využitím obecných výpočtů na čipu grafické karty (GPGPU).

1.1 Cíle práce

Obecně je cílem dosáhnout takové úrovně zrychlení výpočtu, aby byla aplikace využitelná pro řešení reálných propojovacích sítí. Úkolem této práce je analyzovat možnosti akcelerace pomocí paralelního zpracování na GPU a implementovat řešení dosahující maximálního výkonu na zvolené platformě – moderních grafických kartách podporujících obecné výpočty. Hlavní body této práce lze vytyčit následovně:

- Výkonnostní analýza aplikace vytvořené v rámci práce [11].
- Identifikace vhodných částí algoritmu pro paralelní zpracování.
- Návrh paralelního řešení na GPU.

- Odhad dosažitelného zrychlení.
- Implementace navrženého paralelního řešení.
- Návrh a implementace optimalizací zaměřených zejména na efektivní práci s pamětí.
- Vyhodnocení dosažených výsledků – srovnání původní a optimalizované verze.
- Současné využití více výpočetních zařízení v systému.
- Vyhodnocení možností distribuce výpočtu na heterogenních výpočetních systémech.

1.2 Struktura práce

Úvodní kapitoly této práce jsou zaměřeny na teoretický popis oblastí týkající se řešené problematiky. Druhá kapitola se zabývá charakteristikou kolektivních komunikací, podrobněji se věnuje popisu různých typů topologií a definuje základní typy kolektivních komunikací.

Ve třetí kapitole je popsán princip fungování evolučních algoritmů, je prezentováno jejich rozdělení a také jejich typické aplikace.

Ve čtvrté kapitole je charakterizována architektura grafických čipů a pamětí. Práce s pamětmi grafické karty je popsána podle paměťového modelu OpenCL.

V páté kapitole jsou uvedeny výsledky výkonnostní analýzy vzorové aplikace. Dále jsou identifikovány části algoritmu vhodné pro paralelní zpracování a je realizován návrh provedení migrace výpočtu na GPU. V další části této kapitoly se práce věnuje způsobu měření výkonu výpočetních zařízení, na kterých byla aplikace testována. Podle změřeného výkonu a Amdahlova zákona je proveden odhad dosažitelného zrychlení. Jsou zde uvedeny využité nástroje pro měření výkonu a popsána metodika měření.

Šestá kapitola shrnuje proces implementace provedeného návrhu a detailně se věnuje navrženým a realizovaným optimalizacím a práci s pamětí.

Dosažené výsledky jsou přehledným způsobem prezentovány ve formě grafů a tabulek v sedmé kapitole.

Osmá kapitola popisuje rozšíření aplikace na více platforem, které jsou podporovány standardem OpenCL. Toto rozšíření umožňuje distribuci výpočtu mezi všechna dostupná podporovaná zařízení v systému. V této kapitole jsou uvedeny také výsledky testování na více více zařízeních a popsány další možnosti rozšíření aplikace v tomto směru.

V závěru jsou shrnuty výstupy této diplomové práce a popsán její celkový přínos. Dále je diskutován možný budoucí směr rozvoje tohoto projektu.

Kapitola 2

Kolektivní komunikace

Kolektivní komunikace (CC) [11] jsou definovány jako komunikace pravidelné v časové a prostorové rovině realizované mezi paralelně prováděnými procesy. Tento způsob komunikace se označuje jako kolektivní, jelikož se na něm podílí dva a více procesů, které jsou zapojeny do paralelního výpočtu a kolektivně spolupracují při výměně dat. Například se může jednat o výměnu mezivýsledků potřebných pro další krok výpočtu. Tento způsob komunikace představuje významný prostor pro optimalizaci paralelních alogoritmů z hlediska efektivity výměny dat. Ať už se jedná o paralelní výpočty na čipu nebo masivně distribuované výpočty na geograficky vzdálených počítačích, je rychlost a efektivita komunikace mezi jednotlivými výpočetními uzly jedním z rozhodujících faktorů určující celkový výkon aplikace.

Tato práce je zaměřena na optimalizaci kolektivních komunikací na vysokorychlostních systémech na čipu (SoC). Paralelní algoritmy realizované na SoC typicky provádí iterační výpočty, kdy v jedom místě algoritmu je nutné sdílet výsledky s ostatními uzly. Toto místo představuje při špatném naplánování komunikace vysoké riziko zpomalení celého výpočtu. Pokud má být například hodnota na každém procesoru rozptýlena mezi všechny ostatní procesory, je nutné rozeslat celkem p^2 zpráv, kde p je počet procesorů spoluracujících na výpočtu. Optimalizace kolektivních komunikací je proto důležitou součástí většiny paralelních aplikací. V následujících třech podkapitolách jsou popsány různé topologie propojovacích sítí mezi výpočetními uzly, typy kolektivních komunikací a problém přesného plánování kolektivních komunikací.

2.1 Topologie propojovacích sítí

Podle [5] lze využitím propojovacích sítí oproti vyhrazeným komunikačním kanálům dosáhnout mnohem vyššího vytížení komunikačního média. Vyhrazené komunikační kanály jsou totiž po většinu času nevyužité. V propojovacích sítích je komunikace zajištěna pomocí směrování zpráv mezi propojenými uzly. Komunikační kanály jsou v propjovacích sítích sdílené. Konkrétní způsob propojení jednotlivých uzlů je definován zvolenou topologií. Výhodou tohoto řešení oproti výhrazeným linkám je snadnější realizace a zejména pak nižší náklady.

Existují ruzné typy propojovacích sítí, přičemž volba konkrétní topologie je úkolem návrháře systému. Hlavními kritériemi této volby jsou požadavky vyvíjené aplikace na výkon, odezvu a propustnost a náklady na realizaci zvolené topologie. V [11] jsou mezi základní parametry pro vyhodnocení kvality topologie zařazeny šířka bisekce, vytížení kanálu

a zpoždění cesty.

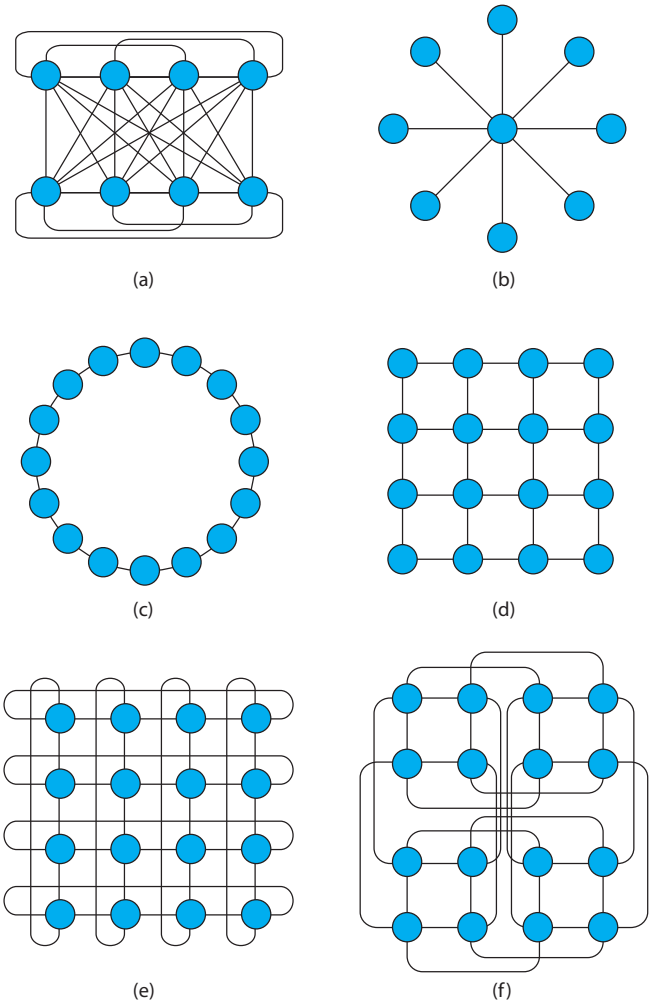
Topologie sestává z množiny výpočetních jednotek a množiny komunikačních kanálů. Pro popis topologie lze využít terminologii teorie grafů. Na množinu výpočetních jednotek lze pohlížet jako na množinu uzlů V, množina komunikačních kanálů odpovídá množině hran C. Každou hranu $c \in C$ lze tedy definovat jako uspořádanou dvojici:

$$c = (x, y)$$
$$x, y \in V$$

2.1.1 Základní vlastnosti topologií

- Průměr – Délka nejdelší z nejkratších cest mezi všemi dvojicemi uzlů.
- Konektivita – Nejmenší počet hran, které při odebrání způsobí rozpad topologie.
- Šířka bisekce – Nejmenší počet hran, které spojují zhruba stejně velké části topologie.

Na obrázku 2.1 jsou znázorněny příklady některých často používaných topologií.



Obrázek 2.1: Znázornění topologií propojovacích sítí: **(a)** Úplné propojení, **(b)** Hvězda, **(c)** Kruh, **(d)** 2D mřížka, **(e)** 2D torus, **(f)** 4D hyperkostka

2.1.2 Cesta

Cesta v topologii mezi dvěma uzly je definována jako uspořádaná n-tice l tvořená posloupností hran $c \in C$ z výchozího uzlu $x \in V$ do cílového uzlu $y \in V$. V každé topologii musí existovat cesta mezi všemi dvojcemi uzlů. Cest mezi uzly x a y může být i více.

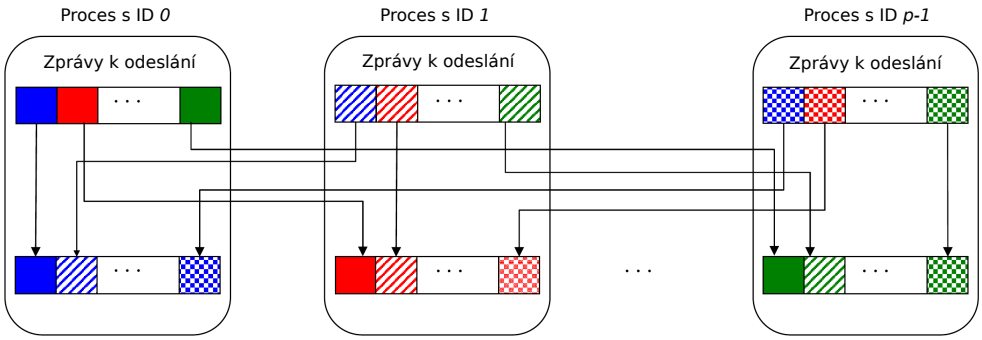
2.2 Typy kolektivních komunikací

Do kolektivních komunikací spadají dva základní typy komunikací [11]. Jedná se o tzv. rozptyl (dále bude využíván anglický termín *scatter*) a rozhlášení (dále bude využíván anglický termín *broadcast*). Tyto dva druhy komunikací lze dále podle počtu účastníků rozdělit na typ *jeden všem* (*one to all*) a *všichni všem* (*all to all*).

2.2.1 Scatter

V rámci tohoto způsobu komunikace pošle procesor p_i zprávy se svými hodnotami všem procesorům (každému procesoru je poslána jedna zpráva s příslušnou hodnotou danou ID procesoru). V tomto případě se jedná o komunikaci typu OAS (*one to all scatter*), při které je rozesláno p zpráv (resp. $p-1$ neuvažujeme-li zprávu sama sobě), kde p je počet procesorů spoluracujících na výpočtu.

Druhou variantou tohoto typu komunikace je AAS (*all to all scatter*), jenž je znázorněna na obrázku 2.2. V tomto případě rozesílají zprávy všechny procesory všem procesorům. Je tedy nutné rozeslat p^2 zpráv.

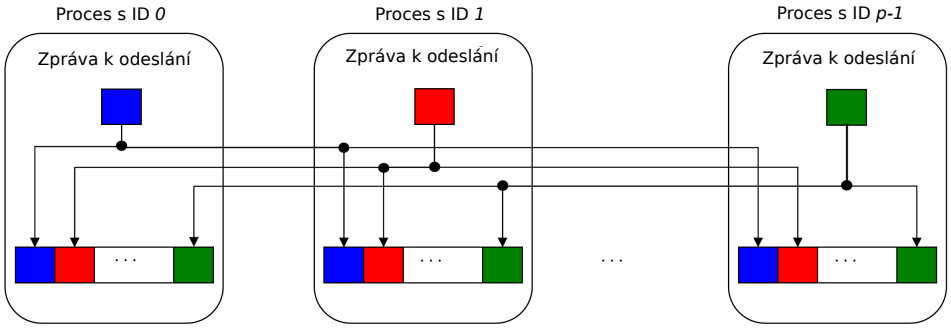


Obrázek 2.2: Schéma kolektivní komunikace typu AAS [11]

2.2.2 Broadcast

U kolektivní komunikace typu *OAB* (*one to all broadcast*) rozhlášuje procesor p_i zprávu se svoji hodnotou. Procesory, které zprávu obdrží se mohou podílet na rozhlášení. Celkově je opět rozesláno p zpráv.

Stejně jako u komunikace *scatter* existuje i zde varianta, kdy svoji hodnotu rozhlašují všechny procesory. Potom se jedná o komunikaci typu *AAB* (*all to all broadcast*), která je znázorněna na obrázku 2.3. V součtu je při komunikaci *AAB* rozesláno opět p^2 zpráv.



Obrázek 2.3: Schéma kolektivní komunikace typu AAB [11]

Kapitola 3

Evoluční algoritmy

Evoluční algoritmy se spolu s neuronovými sítěmi, fuzzy systémy, teorií chaosu a dalšími netradičními přístupy řadí do množiny metod *soft computingu*, který je součástí takzvaného *natural computingu* neboli počítání podle přírody [4]. Tato oblast nachází inspiraci v pozorování lidské společnosti, koloniích interagujících organismů, usuzování jedinců, konfliktech predátor–oběť, systémech na hraně mezi chaosem a řádem apod. [23].

Evoluční algoritmy lze ve všeobecnosti chápat jako abstrakci a formalizaci Darwinovy evoluční teorie. Pomocí evolučních algoritmů je možné numericky simulovat darwinovskou evoluci založenou na přirozeném výběru a náhodném genetickém driftu. Evoluční algoritmy poskytují univerzální algoritmus pro simulaci evoluce, ve kterém je zapotřebí modifikovat pouze fitness chromozómu pomocí odpovídajícího řetězce symbolů (v biologii se tento problém nazývá zobrazení genotypu na fenotyp, kde lineární řetězec chromozómu kóduje organizmus, jehož schopnost reprodukce a přežití je mírou fitness daného chromozómu). [17]

3.1 Základní pojmy evolučních algoritmů

3.1.1 Křížení

Křížení je základním operátorem evolučních algoritmů při vytváření nové generace jedinců. Hlavní uplatnění nachází spolu s operátorem mutace u genetických algoritmů. Křížení probíhá typicky mezi dvěma jedinci a jeho výsledkem jsou dva potomci, jejichž geny vzniknou kombinací genů rodičů. Rozlišují se tři základní druhy křížení:

- Jednobodové – genotyp rodičů je rozdělen a prohozen podle náhodně vygenerovaného bodu.
- Vícebodové – genotyp rodičů je rozdělen a prohozen podle několika náhodně vygenerovaných bodů.
- Uniformní – u každého genu se náhodně rozhodne, od kterého rodiče se zdědí.

3.1.2 Mutace

Operátor mutace vytváří z vybraného jedince mutovaného jedince. V případě binární reprezentace dojde k negaci určitého počtu bitů chromozómu. Pravděpodobnost mutace p_m určuje, s jakou pravděpodobností dojde k mutaci jednoho genu (bitu) chromozómu. Tato

pravděpodobnost se obvykle volí velmi malá, např. $p_m = 0.1\,\%$. [23] Podle [11] jsou nejčastěji používanými typy mutace:

- bitová mutace při binární reprezentaci
- mutace založená na náhodném přenastavení hodnot při celošíselné reprezentaci
- uniformní a neuniformní mutace s pevným rozložením
- permutace založená na operátoru mutace

3.1.3 Fitness funkce

Jedná se o funkci, která jedincům populace přiřazuje hodnotu udávající kvalitu řešení, které jedinec reprezentuje. Jinými slovy, fitness funkce představuje nástroj k hodnocení kvality genotypu jedince.

V případě této aplikace je fitness funkce definovaná jako součet nalezených konfliktů mezi všemi dvojicemi cest naplánovaných na každý časový krok v rámci celého chrozómu. Tato suma je nakonci vynásobena minus jedničkou, nejlepší řešení tedy reprezentuje jedinec s nejvyšší hodnotou fitness. Jedinci představujícímu bezkolizní komunikační plán přiřadí fitness funkce hodnotu nula. V případě nalezení takového jedince je splněna ukončující podmínka algoritmu a výpočet dále nepokračuje. Způsob hledání konfliktů je popsán v kapitole 6.1 a ilustrován obrázkem 6.3.

3.1.4 Princip selekce

Selekce představuje mechanismus, podle kterého jsou vybírání jedinci do nové generace. Selekce se snaží co nejvíce přiblížit přírozenému výběru v přírodě. Jedinci s největší hodnotou fitness by měli být s největší pravděpodobností zvoleni do nové generace, zároveň je však důležité zachovat rozmanitost jedinců v populaci. Nastavením příliš přísných parametrů selekce se do nové populace mohou dostat jedinci reprezentující pouze několik málo byť dobře hodnocených řešení a výpočet může uváznout v lokáním extrému funkce.

Výběr vhodného selekčního mechanismu se odvíjí od konkrétního řešeného problému. Podle [7] jsou nejčastěji využívány tyto metody:

- výběr ruletou
- $(\mu + \lambda)$ a (μ, λ) výběr
- turnaj
- ohodnocování a škálování
- sdílení

Podrobnější popis jednotlivých metod selekce a způsob její realizace v řešené aplikaci uvádí [11]. Testování některých principů selekce a jejich parametrů je prezentováno v [9].

3.2 Typy evolučních algoritmů

V [11] jsou jako nejvýznamnější druhy evolučních algoritmů uvedeny genetické algoritmy, evoluční strategie, evoluční programování a genetické programování. Tyto skupiny mají podle [23] následující společné rysy:

- Používají populaci kandidátních řešení, která umožňuje paralelní přístup k prohledávání.
- K vytváření nových kandidátních řešení používají biologií inspirované operátory, které kombinují kandidátní řešení existující v daném čase.

3.2.1 Genetické algoritmy

Tuto třídu algoritmů představil v roce 1975 jako první John Holland, který se zabýval adaptací v přírodě a v umělých systémech [8]. Podle [18] jsou genetické algoritmy nejbližším výpočetním modelem přirozené evoluce. Dále uvádí, že úspěšnost prohlédávání komplexního nelineárního prostoru a obecná robustnost genetických algoritmů vedly k jejich nasazení při řešení praktických problémů jako je plánování, finanční modelování a optimalizace.

Genetické algoritmy jsou třídou evolučních algoritmů, mají stejnou strukturu a mnoho společných rysů. Z tohoto důvodu bývají často za evoluční algoritmy nesprávně zaměňovány. Jejich základem je populace jedinců (chromozomů) reprezentujících kandidátní řešení. Chromozóm je v [29] definován jako řetězec genů, které představují typicky binárně kódované parametry jedince. Sada těchto parametrů jedince se označuje jako genotyp. Význam genotypu z hlediska vlastností jedince se pak označuje jako fenotyp.

Existuje několik variant genetických algoritmů. Princip fungování jejich zakladní verze lze shrnout do těchto kroků:

1. Vytvoření počáteční generace – náhodně vygenerována populace jedinců.
2. Výpočet hodnoty fitness pro každého jedince v populaci.
3. Vyhodnocení ukončující podmínky – "Je v populaci jedinec s požadovanou hodnotou fitness?"
4. Vytvoření nové generace – princip selekce a aplikace operátorů křížení a mutace.
5. Pokračování 2. krokem.

3.2.2 Evoluční strategie

Současně s vývojem genetických algoritmů vznikal v Evropě v šedesátých letech algoritmus pojmenovaný jako "Evoluční strategie"(ES). Tento algoritmus byl vyvinut na Technické univerzitě v Berlíně výzkumníky P. Bienertem, I. Rechenbergem a H. P. Schwefelem. První aplikace tohoto algoritmu byly zaměřeny na problémy funkčního designu z oblasti strojního inženýrství. [29]

Pro evoluční strategie je typické, že optimalizují vektor reálných parametrů. Základní varianta evoluční strategie používá pouze operátor mutace, který modifikuje každý parametr vektoru. Mutace využívá Gaussovo rozložení s nulovou střední hodnotou a rozptylem σ. Hodnoty vygenerované dle tohoto rozložení jsou přičteny k hodnotám parametrů rodiče. Rovněž parametr σ je modifikován, a to tak, aby zhruba pětina mutací byla úspěšná, tj.

aby pětina potomků byla lepší, než jsou rodiče. Pokud je úspěšných potomků méně, vliv mutací se sníží, pokud více, vliv mutací se zvýší. Tento postup je znám jako "pravidlo jedné pětiny". [23] Evoluční strategie a genetické algoritmy se podle [29] liší hlavně v těchto bodech:

- ES reprezentují jedince v oboru reálných čísel, zatímco GA využívají hlavně binární reprezentaci.
- ES většinou nepoužívají operátor křížení, ale jen operátory selekce a mutace.

Mezi typické příklady evolučních strategií patří:

- Dvoučlenné ES
- Vícečlenné ES
- Rekombinační ES
- Adaptivní ES

Podrobný popis evolučních strategií, charakteristiku jednotlivých skupin a jejich aplikace uvádí Beyer v [3].

3.2.3 Evoluční programování

Přestože se evoluční programování vyvíjelo prakticky nezávisle od evolučních strategií, obsahují tyto dva přístupy mnoho společných prvků. Pojem evolučního programování zavedl jako první Lawrence Fogel v šedesátých letech 20. století v USA. Evoluční programování typicky používá aplikačně specifickou reprezentaci problémů, samoadaptaci a turnajovou selekci. Křížení se většinou nepoužívá. [23]

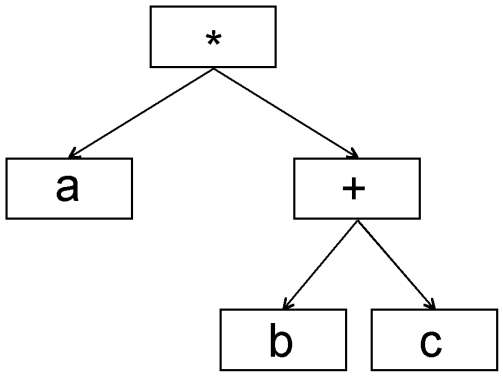
3.2.4 Genetické programování

Tato třída evolučních algoritmů je zaměřena na vývoj počítačových programů pomocí principů založených na přirozené evoluci. Populace je tvořena jedinci, kteří reprezentují počítačové programy. Genetické programování spočívá v nalezení počítačového programu s nejlepší hodnotou fitness z prostoru všech možných programů vytvořených z daných množin F (množina funkcí) a T (množina terminálů). Kódování programů se provádí zejména pomocí stromu, jehož uzly jsou prvky z množiny $V = F \cup T$. Způsob zakódování funkce $a * (b + c)$ je znázorněn na obrázku 3.1.

Využívá se křížení i mutace. Při křížení dvou jedinců vznikají potomci výměnou typicky dvou náhodně vybraných podstromů rodičů. Mutace způsobí náhradu podstromu jedince náhodně vygenerovaným podstromem. Podrobněji se tímto přistupem zabývá John R. Koza v [16].

3.3 Typické aplikace evolučních algoritmů

Mezi hlavní oblasti aplikací evolučních algoritmů patří optimalizační problémy. Jedná se zejména o problémy, které nelze řešit analyticky nebo by jejich analytické řešení bylo příliš komplikované. Jako konkrétní aplikace lze uvést například nalezení optimální trajektorie robota, aerodynamická optimalizace geometrie křídla [29], umělá inteligence ve hře šachy



Obrázek 3.1: Ukázka stromové reprezentace matematických výrazů v genetickém programování.

[24] nebo také evoluční návrh kolektivních komunikací [11]. V [22] je kromě optimalizačních problémů mezi další aplikace evolučních algoritmů zařazeno modelování, strojové učení a umělá inteligence, plánování a řízení či dolování dat.

Evoluční algoritmy s populací jedinců reprezentujících kandidátní řešení se ukázaly jako nejvhodnější přístup k problematice optimalizace a plánování kolektivních komunikací. Tento přístup je využit také v evolučním návrhu, jenž je základem aplikace, kterou tato práce analyzuje a navrhuje způsob dosažení akcelerace pomocí GPGPU. Analýza algoritmu a výběr částí pro realizaci na GPU jsou popsány v kapitole 5.

Kapitola 4

Architektura GPU

GPU, čip na grafické kartě, sloužil od počátku pouze k urychlení vykreslování obrazu na výstupním zobrazovacím zařízením počítače. Jeho smyslem bylo jen odlehčení zátěže procesoru. Tato skutečnost však přestává platit počátkem 21. století, kdy je v GPU objeven obrovský výpočetní potenciál také pro obecné typy výpočtů. V současné době se tato oblast stále rozvíjí a pro GPU se nachází různá uplatnění, například na poli fyzikálních simulací, bioinformatiky nebo zpracování obrazových a zvukových signálů.

Tato kapitola se zabývá architekturou grafického čipu, jejím srovnáním s architekturou klasického procesorem a podrobněji popisuje paměťovou hierarchii grafických karet. Jsou uvedeny charakteristické vlastnosti jednotlivých typů pamětí a dále je popsán způsob práce s těmito pamětmi.

Svou architekturou a způsobem provedení se grafický čip řadí do skupiny označované jako SIMD (Single Instruction, Multiple Data). To znamená, že umožňuje provádění jedné instrukce současně nad různými daty. GPU je obecně tvořen z velkého počtu výpočetních jader, které jsou po osmi uspořádány do multiprocesorů. Každý multiprocesor má k dispozici vlastní registry, lokální vyrovnávací paměť, paměť konstant a paměť pro textury. Grafická karta obsahuje také globální paměť, která je sdílená pro všechny multiprocesory. Schéma uspořádání jednotlivých pamětí a jejich popis je uveden v části 4.2.

Rychlost vypočetních jader u GPU je sice nižší než u CPU, jejich síla však spočívá v jejich počtu. Každý problém, který má být na GPU efektivně řešen, musí být nejprve dekomponován a upraven tak, aby jej bylo možné dobře paralelizovat. Pro dosažení maximálního zrychlení je totiž nutné, aby byly pokud možno všechny výpočetní jednotky neustále vytíženy. To znamená rozdělit běh programu na několik tisíc nezavislých vláken, které mohou běžet paralelně.

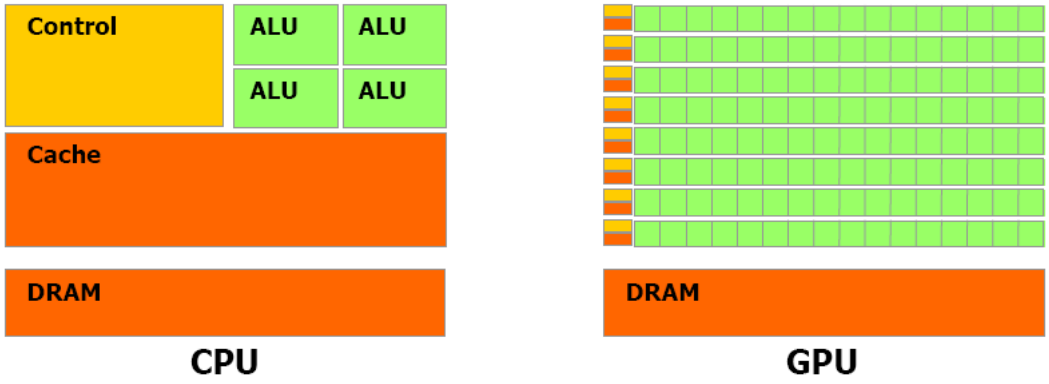
Všechna vlákna zpracovávaná na GPU jsou rozdělena do skupin po 32 vláknech na tzv. *warpy*, které jsou spouštěny na multiprocesorech. To znamená, že každý multiprocesor je schopen v jednom okamžiku obsluhovat více vláken, aniž by mezi nimi musel přepínat. Nevýhodou tohoto hromadného zpracování je přítomnost pouze jednoho IP (Instruction Pointer) registru pro každý warp. To s sebou přináší omezení, že všechna vlákna v jednom warpu vykonávají vždy stejnou instrukci.

4.1 Porovnání s CPU

Na obrázku 4.1 je zjednodušené schéma demonstrující hlavní rozdíly mezi procesorem a grafickým čipem. Hlavní předností CPU je velmi rychlá a velká vyrovnávací paměť, vysoká

výkonnost při vykonávání jednoho vlákna, podpora vstupně vystupních operací a také kvalitní predikce skoků. Naproti tomu spočívají hlavní přednosti GPU v obrovském počtu výpočetních jader a rychlých lokálních pamětech.

Současné čipy GPU nabízejí vysoký výpočetní potenciál. Nejnovější grafický čip od společnosti nVidia s názvem GeForce GTX 690 obsahuje 3072 vypočetních jader[1], což představuje oproti současným maximálně šestijádrovým CPU obrovský rozdíl. Pokud však implementací na GPU chceme dosáhnout znatelného zrychlení, musíme nejprve provést důkladnou analýzu implementovaného problému a zvážit, jestli je daný problém dobře paralelizovatelný a tedy vhodný pro zpracování na GPU.



Obrázek 4.1: Porovnání architektury CPU a GPU [20]

4.2 Popis pamětí grafické karty

Vedle hierarchie pamětí celého počítače, která je tvořena různými typy pamětí od pevných disků, přes operační paměť, vyrovnávací paměť až po registry procesoru, existuje obdobná paměťová hierarchie v rámci grafické karty. I zde samozřejmě platí, že čím blíže k výpočetním jádrům, tím rychlejší avšak kapacitně menší a nákladnější paměť. Každý standard jako je OpenCL nebo CUDA umožňující obecné výpočty na GPU definuje vlastní paměťový model. Modely musí odpovídat fyzickým pamětem v grafické kartě, tudíž se liší zejména označením jednotlivých paměťových prostorů a metodami přístupu. Aplikace vytvářená v rámci této práce je implementována v C++ a OpenCL, z tohoto důvodu bude popis pamětí a práce s nimi uveden z pohledu OpenCL.

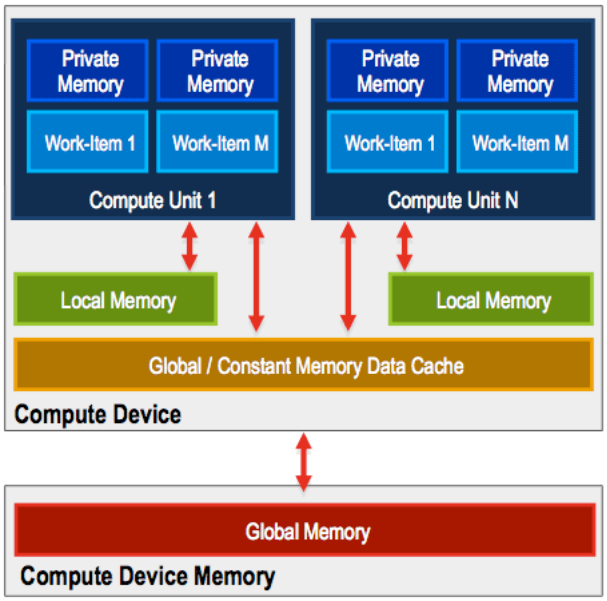
Paměťový model použitý v OpenCL se označuje výrazem *relaxed consistency model* a je popsán ve specifikaci OpenCL. Tento model je chrakterizován následujícími body [6]:

- Umožňuje vláknům přistupovat do jejich privátní paměti.
- Dovoluje sdílet lokální paměť mezi vlákny v rámci jednoho bloku. Konzistence je však zaručena pouze při použití různých synchronizačních nástrojů, například bariér.
- Různé bloky mezi sebou nemohou komunikovat ani se navzájem nijak synchronizovat. Konzistence dat mezi různými pracovními skupinami není zajištěna.

[1] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications

- Datové závislosti mohou být definovány a zajištěny pomocí pracovních front a atomických operací.

Na obrázku 4.2 je znázorněno schéma paměťové hierarchie grafické karty odpovídající paměťovému modelu OpenCL. Tento model rozděluje paměť na čtyři základní typy – globální paměť, paměť konstant, lokální paměť a privátní paměť. Tyto paměti se liší zejména kapacitou, dobou přístupu a rychlostmi operací zápisu a čtení. Dále je paměti možné rozdělit podle toho, komu jsou přístupné. Charakteristika jednotlivých typů pamětí grafické karty je podrobně popsána ve specifikaci OpenCL [14]. Na základě [6] a specifikace OpenCL uvádí následující podkapitoly hlavní vlastnosti jednotlivých typů pamětí.



Obrázek 4.2: Paměti grafické karty z pohledu OpenCL [6]

4.2.1 Globální paměť

Globální paměť je na grafické kartě z hlediska kapacity obecně největší pamětí, zároveň však také nejpomalejší. Současné moderní grafické karty jsou vybaveny globální pamětí v řadech gigabajtů. Šířka přenosového pásma u tohoto typu pamětí sice umožňuje dosáhnout přenosové rychlosti okolo 200 GB/s ovšem zpoždění při přístupu do této paměti dosahuje hodnot 400 až 600 hodinových cyklů [20]. Tento paměťový prostor je přístupný pro čtení i zápis všem vláknům ve všech blocích. Vlákna mohou číst ze všech položek paměťových objektů uložených v této paměti nebo do nich zapisovat. Zapisovaná a čtená data mohou být v závislosti na možnostech výpočetního zařízení ukládána do vyrovnávací paměti.

4.2.2 Paměť konstant

Jedná se typicky o část globální paměti[2], která po celou dobu vykonávání kernelu zůstává neměnná. Paměťové objekty v této paměti musí být alokovány a inicializovány z hostitelské aplikace před spuštěním kernelu.

[2] záleží na daném zařízení a výrobci

4.2.3 Lokální paměť

Tento paměťový prostor je lokální vzhledem k jednotlivým blokům spuštěným v rámci provádění kernelu. Lokální paměť bývá umístěna přímo na GPU čipu a doba přístupu je tak mnohem kratší, než do globální paměti. Velikost této paměti lze na každém zařízení zjistit voláním funkce `clGetDeviceInfo()` s parametrem `CL_DEVICE_LOCAL_MEM_SIZE`. Díky tomu lze i za běhu aplikace upravit velikost alokované paměti nebo počet spouštěných bloků tak, aby se minimalizovaly přístupy do globální paměti.

4.2.4 Privátní paměť

Jedná se o paměťový prostor vyhrazený každému běžícímu vláknu. Proměnné definované v privátní paměti jednoho vlákna nejsou viditelné žádnému jinému vláknu. Svými vlastnostmi se tato paměť nejvíce blíží registrům procesoru. Operace čtení a zápisu tak nevyžadují žadný synchronizační mechanismus a podle [20] při přístupu do privátní paměti nevzniká žádné zpoždění. OpenCL standard nedefinuje velikost ani umístění privátní paměti, tyto parametry se odvíjí od konkrétního zařízení. V případě, že je alokováno příliš velké množství této paměti, může být paměť mapována do pomalejších typů pamětí jako je například globální paměťový prostor. Vzhledem k rozdílu v přístupových dobách do privátní a globální paměti může mít tato skutečnost markantní dopad na výkon aplikace. Pro kontrolu, do kterých pamětí vlákna kernelu přistupují, lze použít nVidia Visual Profiler[3].

4.2.5 Rozdělení přístupu

Vzhledem k paměťovému modelu a implementaci jednotlivých typů pamětí do nich nelze zcela libovolně přistupovat. Podle [14] je v tabulce 4.1 uvedeno rozdělení jednotlivých typů pamětí na základě možností přístupu a způsobu alokace.

Tabulka 4.1: Možnosti alokace a přístupu do pamětí grafické karty

	Globální paměť	Paměť konstant	Lokální paměť	Privátní paměť
Hostitel	dynamická alokace čtení i zápis	dynamická alokace čtení i zápis	dynamická alokace žádný přístup	– žádný přístup
Kernel	– čtení i zápis	statická alokace pouze čtení	statická alokace čtení i zápis	statická alokace čtení i zápis

Pro vymezení typu proměnné se používají následující kvalifikátory:

- `__global`
- `__constant`
- `__local`
- `__private`

První tři se typicky vyskytují u argumentů kernelových funkcí. Proměnné primitivních datových typů definované uvnitř kernelu jsou automaticky privátní, při nedostatku paměti nebo v případě polí jsou data uložena v oblasti lokální paměti.

[3] http://developer.nvidia.com/nvidia-visual-profiler

Kapitola 5

Akcelerace výpočtů pomocí GPU

Cílem této práce není vývoj nového evolučního algoritmu pro plánování kolektivních komunikací, ale akcelerace existujícího řešení. Vzorem, ze kterého práce vychází, je aplikace implementující evoluční algoritmus vytvořená doktorem Jarošem v rámci jeho disertační práce [11]. Tato práce tedy vychází z funkčního a otestovaného řešení, které navrhuje akcelerovat paralelním zpracováním na GPU. Vzorová aplikace byla analyzována z hlediska časové náročnosti výpočtu. Získané výsledky z analýzy jsou prezentovány v této kapitole spolu s návrhem řešení a odhadem dosažitelného zrychlení.

Při hledání komunikačního plánu u složitějších topologií se čas potřebný ná výpočet dostává za rozumnou hranici. Tabulka 5.1 uvádí průměrný čas potřebný k nalezení komunikačního plánu na procesoru Intel Core 2 Duo 2.8 Ghz pro poměrně malé topologie[1].

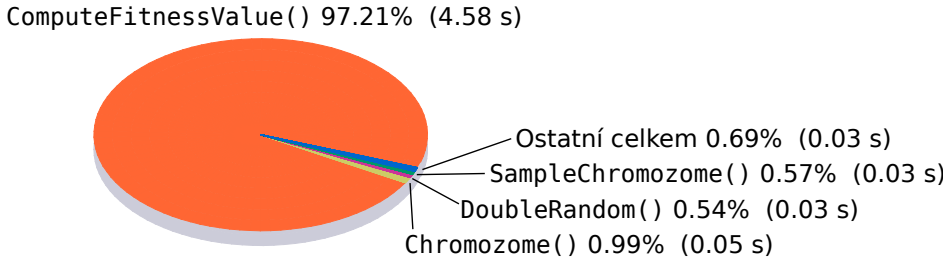
Tabulka 5.1: Čas potřebný k nalezení komunikačního plánu

Topologie	Uzly	Časové kroky	Čas výpočtu	Počet generací
Mřížka 2x2	4	2	0.011 s	5
Mřížka 3x3	9	6	8.932 s	6436
Mřížka 4x4	16	19	53.809 s	7373

Z tabulky 5.1 vyplývá, že čas potřebný pro nalezení bezkolizního komunikačního plánu se zvětšujícím se počtem procesorů v topologii strmě nárůstá. Nalezení bezkolizního komunikačního plánu představuje NP-úplný problém a doba výpočtu se tedy zvyšuje exponenciálně. S rostoucí velikostí topologie se zároveň zvyšuje i doba potřebná pro ohodnocení jednoho jedince. Tato doba je dána $n^2 \cdot d^2$, kde n je počet uzlů a d je průměrná délka nejkratších cest. Řešení rozsáhlejších topologií je proto časově velmi náročné. Nasazení tohoto evolučního algoritmu pro hledání optimálních plánů reálných topologií je možné pouze při dosažení markantního zrychlení výpočtu. Za účelem identifikace časově nejnáročnějších částí algoritmu byla provedena analýza aplikace pomocí profilačního programu *gprof*. Výstupy profilace jsou znázorněny v grafu 5.1.

Výsledky profilace ukázaly, že nejvíce času (přes 97 %) program stráví výpočtem fitness jedinců neboli ohodnocením kvality kandidátních řešení. Hodnotící funkce fitness je ústřední částí každého evolučního algoritmu. Pokud má být docíleno akcelerace algoritmu, je nutné zaměřit se právě na tuto funkci.

[1] Parametry výpočtu: velikost populace = 100, pravděpodobnost mutace = 0.7, počet vláken = 2



Obrázek 5.1: Výsledky profilace aplikace programem gprof. Patametry testu: 1000 generací, topologie mřížka 7x7, 200 jedinců v populaci. Časové údaje odpovídají řešení jedné generace na procesoru Intel Core i7 920.

Výpočet hodnoty fitness jedinců nové populace je založen na volání funkce `computeFitnessValue()`. Jinými slovy, můžeme říci, že totožný kód se provede tolikrát, jaký je počet nových jedinců v populaci. Vzhledem ke skutečnosti, že velikost populace se typicky pohybuje v řádu stovek jedinců, nabízí se jako vhodné řešení dosáhnout akcelerace migrací tohoto výpočtu z klasického CPU na architekturu typu *SIMD* obsahující velké množství výpočetních jader. Konkrétním příkladem této architektury jsou moderní grafické karty.

5.1 Návrh implementace na GPU

Návrh byl proveden na základě profilace vzorové aplikace. Spočívá v zachování provedení většiny kódu evolučního algoritmu na klasickém CPU a pouze výpočet hodnoty fitness jedinců v populaci bude probíhat na GPU. Jednoduché schéma na obrázku 5.2 znázorňuje algoritmus zahrnující provedení části výpočtu na GPU.

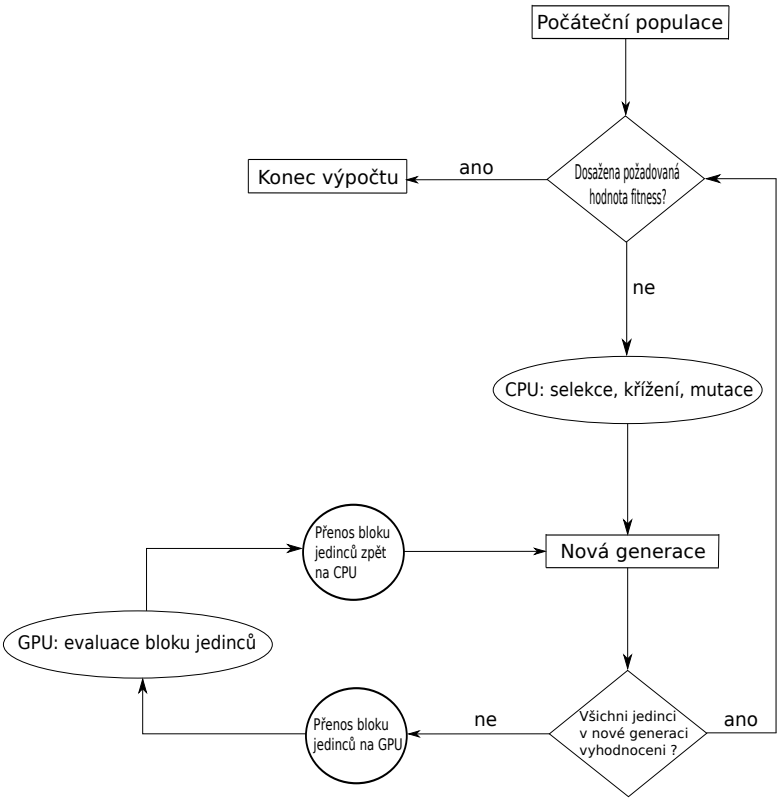
Míra zrychlení dosažená pomocí paralelního výpočtu na GPU může být značně snížena režií, například neoptimalizovanými paměťovými přenosy. Z tohoto důvodu bylo nutné vytvořit vhodný návrh pro přenos jedinců k ohodnocení mezi hostitelskou pamětí počítače a pamětí GPU. Mezi možné způsoby realizace lze zařadit následující postupy:

- Přenos a ohodnocení každého jedince zvlášť
- Rozdělení populace na více částí a zpracování po skupinách
- Přenos a ohodnocení všech jedinců současně (bez možnosti překrytí komunikace)

Pokud chceme dosáhnout maximálního zrychlení na architektuře *SIMD*, je nutné zajistit využití nejlépe všech dostupných výpočetních jader současně. Předpokládáme-li komunikační schéma AAS (všichni všem rozptyl) musí každý chromozóm obsahovat gen pro každou dvojici uzlů v topologii. Každý gen pak sestává z indexu cesty a časového kroku. Celková velikost každého jedince je potom dána vzorcem

$$s = 2n^2,$$

kde n je počet uzlů a s velikost jedince (pro získanání velikosti v bajtech nutno vynásobit čtyřmi).



Obrázek 5.2: Schéma návrhu výpočtu pomocí GPU

U běžných topologií, které jsou tvořeny desítkami uzlů, se tedy velikost chromozómu pohybuje v řádu stovek až tisíců bajtů. To znamená, že použití prvního navrhovaného přístupu, přenos a ohodnocení každého jedince zvlášť, by dostatečně nevyužilo výpočetní potenciál GPU čipu a nebylo by dosaženo maximální akcelerace [15]. Naopak přenos a ohodnocení všech jedinců současně neumožňuje překrytí výpočtu a komunikace, zároveň by při velké populaci mohl nastat problém s nedostatkem paměti GPU. Z těchto důvodů se jako nejefektivnější přístup nabízí rozdělení populace na více částí a její zpracování po skupinách. Velikost skupiny bude nastavena konstantou, jejíž hodnotu lze přizpůsobit velikosti a typu řešené topologie a také použitému GPU.

Dalším důležitým prvkem návrhu je způsob zpracování jedinců, kteří se již nacházejí v paměti GPU. Základním prvkem paralelního zpracování na GPU je vlákno (thread). Vlákna jsou rozdělena do skupin označovaných pojmem *warp*, které jsou vykonávány na jednotlivých procesorech. Z programátorského hlediska se pro dosažení větší granularity využívá pojem blok, jehož velikost (počet vláken, kterými je tvořen) lze nastavit.

Možnosti zpracování jedince jsou následující:

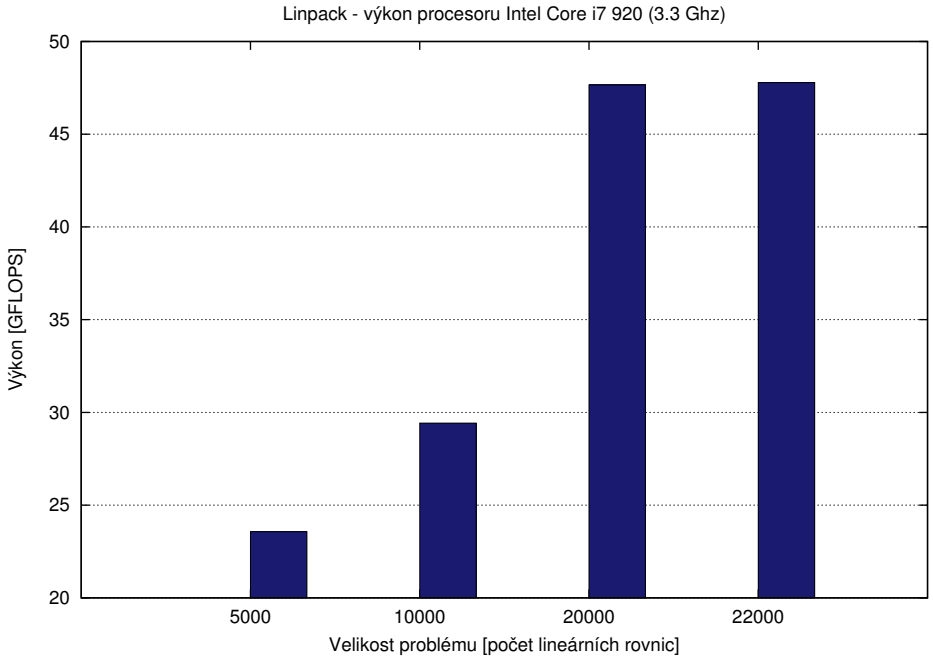
- Přiřazení jedince na blok
- Přiřazení jedince na vlákno
- Rozdělení jedince na části a jejich zpracování více vlákny

V pruběhu vývoje aplikace budou tyto možnosti otestovány, na základě výsledků testů bude zvoleno finální provedení včetně nastavení optimální velikosti bloku.

5.2 Měření výkonu

5.2.1 Měření výkonu CPU

Pomocí nástroje Linpack[2] bylo provedeno deset testů pro různé velikosti problému. Testování bylo realizováno na procesoru Intel Core i7 pracujícím na frekvenci 3.3 GHz. Jedná se o čtyřjádrový procesor s technolgií HyperThreading. Počet vláken při testování byl proto nastaven na osm. Získané výsledky byly zprůměrovány a jsou názorně reprezentovány v grafu 5.3.



Obrázek 5.3: Výsledky testování výkonu CPU nástrojem Linpack

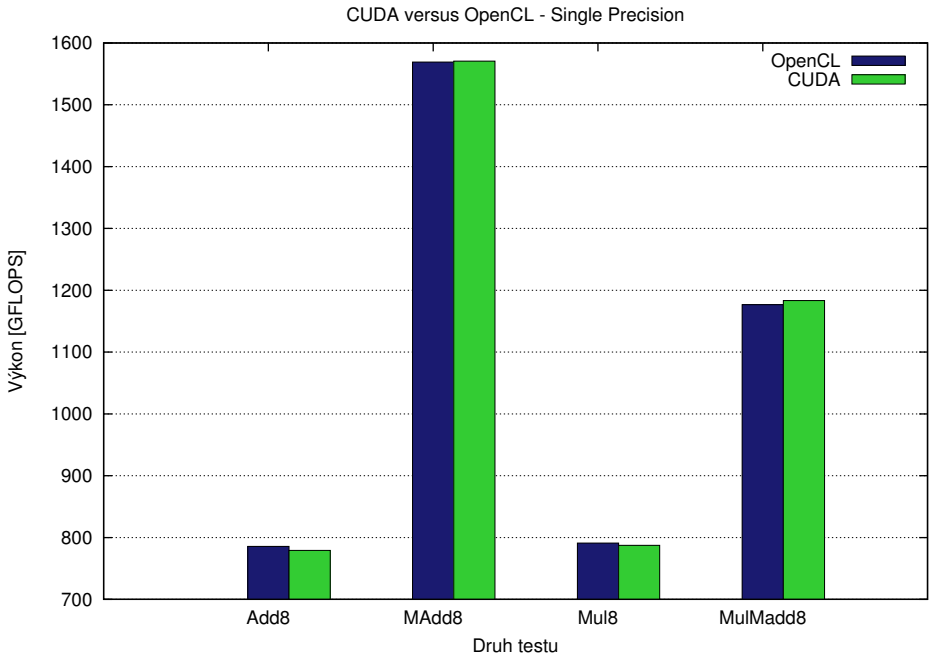
5.2.2 Měření výkonu GPU

Nástrojem SHOC[3] byla otestována grafická karta GeForce GTX 580. Byl použit test pro měření výkonu pro různé aritmetické operace. Testování bylo provedeno s technolgií CUDA i OpenCL. Získané výsledky jsou vyneseny v grafu 5.3. Dosažený výkon se mezi porovnávanými technologiemi v jednotlivých testech mírně liší. Rozdíly jsou však příliš malé na to, aby na jejich základě mohly být tyto technologie hodnoceny. Rozdíly mohou být způsobeny také různou implementací testovacích funkcí nástroje SHOC.

5.3 Teoreticky dosažitelné zrychlení

V rámci návrhu byl proveden odhad dosažitelného zrychlení převedením funkce `computeFitnessValue()` k paralelnímu zpracování na GPU. K tomuto odhadu byl využit Amdahlův zákon, který je reprezentován vzorcem (5.1). Na základě poměru výkonu mezi

[2] http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download/
[3] https://github.com/spaffy/shoc/wiki



Obrázek 5.4: Srovnání výkonu CUDA a OpenCL v jednoduché přesnosti.

GPU a CPU, jenž byl změřen v části 5.2, a díky znalosti velikosti paralelizovatelné části programu identifikované programem gprof (viz 5.1) je pomocí Amdahlova zákona možné vypočítat maximální teoretické zrychlení.

$$S(P) = \frac{P}{1 + \alpha(P - 1)} \tag{5.1}$$

Hodnota P v uvedeném vzorci vyjadřuje počet procesorů využitých pro paralelizaci. V tomto případě využijeme výkonnostní poměr mezi zvoleným GPU a CPU. Tento poměr vyjadřuje přibližný počet procesorů, jejichž výkon odpovídá zvolenému GPU.

$$P = \frac{\text{výkon GPU}}{\text{výkon CPU}} = \frac{1570}{48} \approx 33$$

Za hodnotu α je dosazena procentuální časová náročnost sekvenčně zpracovávané části aplikace. V tomto případě získáme sekvenční část sečtením všech hodnot v grafu 5.1 výjma hodnoty `computeFitnessValue()`. Sekvenční část této aplikace tvoří 2.79 % z celkové doby výpočtu. Podle Amdahlova zákona je tak teoreticky možné dosáhnout maximálně nasledujícího zrychlení:

$$S(GPU) = \frac{33}{1 + 0.0279 \cdot (33 - 1)} = 17.43$$

Tato hodnota zrychlení je pouze teoretická. Nezahrnuje totiž nutnou režii spojenou s inicializací paralelní části výpočtu. V tomto ohledu je nutné uvažovat režii při vytváření vláken, dále čas potřebný pro přenesení vstupních dat po sběrnici PCI Express do paměti grafické karty a po dokončení výpočtu přenesení výsledků stejným způsobem do operační paměti počítače.

Dosažitelné zrychlení bude dále negativně ovlivněno tím, že výpočetní jádra GPU nejsou univerzálními procesory, ale patří do architektury SIMD. Narozdíl od univerzálních CPU, kde má každý procesor resp. jádro vlastní čítač instrukcí, jsou vlákna na GPU rozdělena do warpů. Velikost warpu je na současných GPU typicky 32 vláken, přičemž všechna vlákna v jednom warpu sdílí jeden čítač instrukcí. Tento způsob provedení se projeví snížením výkonu pokud v rámci warpu dojde k tzv. divergenci vláken. K divergenci dochází při podmíněných skocích a větvení programu, kdy vlákna v jednom warpu provádějí různou část kódu. Vzhledem k povaze řešeného problému, tedy porovnání cest různých délek, kdy dochází k větvení na základě nalezení nebo nenalezení konfliktu, je předpokládána poměrně vysoká úroveň divergence.

Odhad reálně dosažitelného zrychlení se tak při zahrnutí popsaných omezení pohybuje v případě grafické karty nVidia GTX 580 v porovnání s procesorem Intel Core i7 920 okolo osminásobného zrychlení v případě použití grafické karty.

Kapitola 6

Proces implementace

První verze aplikace byla vytvořena zejména s ohledem na funkčnost bez důrazu na optimalizace při práci s GPU. Jejím hlavním účelem bylo vytvořit rozhraní pro komunikaci a přenos dat mezi částí aplikace zpracovávané na CPU a částí na GPU. Následující text popisuje proces implementace této prvotní verze. Navržené a implementované optimalizace jsou podrobně popsány v části 6.2.

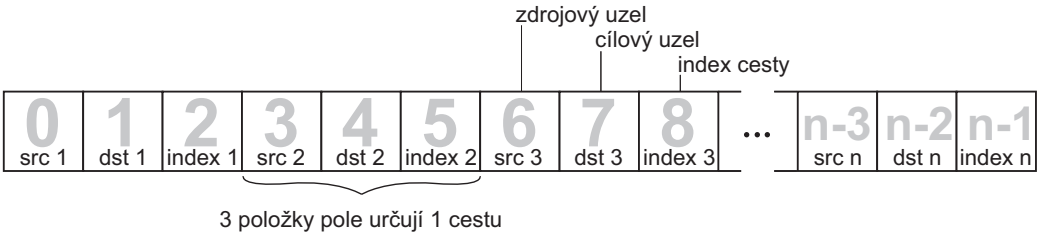
Byl vytvořen nový modul s názvem `GPU.cpp` a příslušný hlavičkový soubor `GPU.h` pro integraci do existující aplikace. Z tohoto modulu může hostitelská část aplikace zavolat tři funkce: `initGPU()`, `launchGPU()` a `cleanUp()`.

Ve funkci `initGPU()` jsou provedeny základní operace pro získání přístupu ke grafické kartě a nastavení parametrů výpočtu. Dále je v této funkci alokována paměť v globálním paměťovém prostoru grafické karty pro uložení pole obsahujícího všechny nejkratší cesty mezi vysílajícími a přijímacími procesory. V hostitelské aplikaci jsou tato data uložena ve čtyřdimenzionálním poli ve formátu `[zdroj][cíl][index cesty][cesta]`. Vzhledem k větší efektivitě při práci s lineárními poli na straně GPU bylo toto pole převedeno na jednodimenzionální. Ve funkci `initGPU()` je tedy alokováno jednodimenzionální pole o velikosti `počet_zdrojů * počet_cílů * maximalní_počet_cest * nejdelší_cesta`. Do tohoto pole jsou uloženy nejkratší cesty mezi každou dvojicí vysílač–přijímač a následně překopírovány pomocí funkce clEnqueueWriteBuffer() do globální paměti grafické karty. Časová a paměťová náročnost operací prováděných ve funkci `initGPU()` se odvíjí od velikosti zvolené topologie a parametrů pro evoluční algoritmus, zejména se jedná o parametry specifikující počet časových kroků a velikost populace. Funkce `initGPU()` je v rámci celého běhu programu volána pouze jedenkrát a to na jeho začátku. Z tohoto důvodu je podíl doby strávené ve funkci `initGPU()` z celkového času běhu aplikace prakticky zanedbatelný. To platí zvláště pro delší běhy aplikace, které jsou pro řešení složitějších topologií typické.

Funkce `launchGPU()` již spouští samotný proces výpočtu hodnoty fitness každého chromozómu. Na základě způsobu, jakým je počítána hodnota fitness, který je popsán v 3.1.3, je možné tento algoritmus rozdělit na dva logické celky. V první části je nalezena množina cest, obsahující cesty mezi vysílajícími a přijímacími uzly naplánované na jeden časový krok. V rámci druhé časti algoritmu jsou cesty z této množiny vzájemně porovnány a jsou nalezeny a sečteny konflikty.

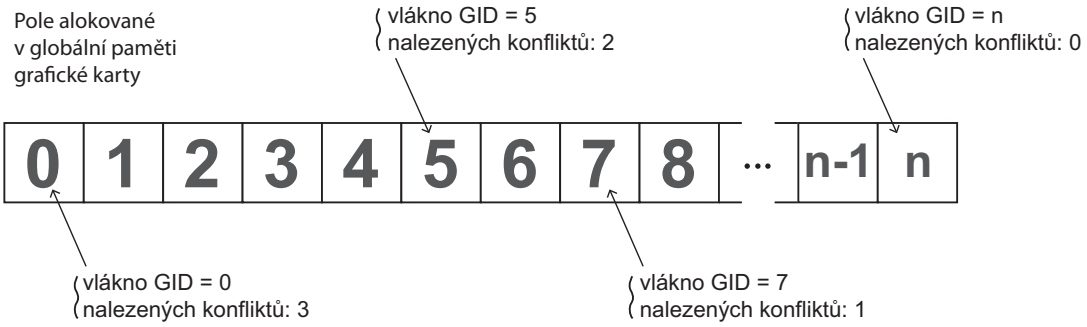
Zpracování první části algoritmu bylo ponecháno v režii CPU ve funkci `MMSComputeFitnessValue()`. V této funkci jsou do jednodimenzionálního pole typu `int` s názvem `sameStepPaths` uloženy informace o cestách naplánovaných na stejný časový krok. Formát uložení dat v tomto poli je znázorněn na obrázku 6.1.

Následně je pole předáno do funkce `launchGPU()`, kde je provedena alokace paměti v glo-



Obrázek 6.1: Způsob uložení identifikace cest v poli `sameStepPaths`

bálním paměťovém prostoru grafické karty a pole `sameStepPaths` je do tohoto paměťového prostoru nakopírováno. Druhá část algoritmu provádějící vlastní porovnání cest naplánovaných na stejný časový krok je realizována v OpenCL. Tento kód je umístěn v souboru `fitness.cl` a je paralelně vykonáván jednotlivými vlákny na výpočetních jádrech GPU. Funkce `launchGPU()` tedy provádí operace nutné pro zahajení výpočtu na GPU. Zajišťuje přenos dat mezi operační pamětí a pamětí grafické karty a detekuje vznik případných chyb. Po dokončení výpočtu na GPU jsou nalezené konflikty uloženy do pole a překopírovány zpět do operační paměti počítače. Způsob uložení výstupních dat reprezentuje obrázek 6.2.



Obrázek 6.2: Zápis dat do výstupního pole v globální paměti

Suma získaná z výsledných hodnot z GPU představuje celkový počet konfliktů pro daný chromozóm a časový krok. Tato hodnota je vrácena do funkce `MMSComputeFitnessValue()` a celý postup je opakován pro všechny časové kroky a chromozómy.

Poslední funkcí implementovanou v modulu `GPU.cpp`, kterou je možné volat z ostatních modulů aplikace, je funkce `cleanUp()`. Výsoký počet dynamických paměťových alokací v kombinaci s mnoha úseky kódu, ve kterých může program neočekávaně skončit, si vynutil implementaci funkce, která umožňuje efektivně uvolnit veškerou dynamicky naalokovanou paměť a v případě nutnosti také celý program ukončit.

Za účelem kontroly výsledků bylo nutné provést úpravy v obou aplikacích. Program využívá při generování jedinců pseudonáhodný generátor, jehož *seed* je nastaven podle aktuálního času a čísla procesu. Každý běh aplikace je tak i při nastavení stejných parametrů evolučního algoritmu jedinečný. Porovnání výsledků a ověření správnosti tak bylo možné provést až po nastavení hodnoty *seed* v obou verzích aplikace na stejnou konstantu. Dalším faktorem, který bylo nutné dočasně eliminovat, protože nepredikovataleným způsobem ovlivňoval výsledné hodnoty, byla mutace jedinců při vytváření nových generací. Tato implementace byla otestována na několika grafických kartách, přičemž výsledky testů byly

kontrolovány proti výsledkům naměřených na čisté CPU implementaci. Testováním bylo ověřeno, že se jedná o funkční implementaci. Testování rychlosti běhu aplikace potvrdilo nutnost zavedení zásadních optimalizací pro zlepšení výkonu. Popis jednotlivých optimalizací, aplikovaných na tuto základní verzi aplikace, je uveden v sekci 6.2.

6.1 Základní algoritmus vyhledávání konfliktů

Tento algoritmus je implementován v jazyce OpenCL C. Jedná se o jazyk C s rozšířeními pro paralelní zpracování. Celá implementace je realizována jedinou funkcí `__kernel void fitness()` a je umístěna v souboru `fitness.cl`. Tato část kódu je zpracována paralelně na GPU, přičemž počet vláken provádějící výpočet je roven hodnotě o jedničku menší, než je počet cest naplánovaných na stejný časový krok.

Pro nalezení všech konfliktů je nutné porovnat každý segment každé cesty se všemi segmenty všech ostatních cest. Současně platí, že se provádí jen jednostranné porovnání, tedy provnáme-li segmenty cesty A se segmenty cesty B, již neprovádíme porovnání cesty B s cestou A. Porovnání je znázorněno na obrázku 6.3.

Uvažujeme-li n cest, přičemž každá cesta má obecně různou délku m_i, lze počet všech nutných operací porovnání vyjádřit jako součin sumy všech dvojic cest a sumy operací porovnání v rámci jedné dvojice cest. Počet všech dvojic cest je dán vzorcem:

$$\frac{n \cdot (n-1)}{2} \tag{6.1}$$

Vztah pro počet porovnání v rámci jedné dvojice cest lze odvodit ze způsobu porovnávání cest. Uvedená délka m_i reprezentuje přesněji počet uzlů, kterými cesta prochází (včetně počátečního a koncového). Skutečná délka cesty je tedy $m_i - 1$. Potřebujeme porovnat každý segment cesty s každým, přičemž porovnání jednoho segmentu vyžaduje dvě operace porovnání – počáteční uzel prvního segmentu s počátečním uzlem druhého segmentu a koncový uzel prvního segmentu s koncovým uzlem druhého segmentu. Počet porovnání v rámci jedné dvojice cest tedy odpovídá hodnotě dané vztahem:

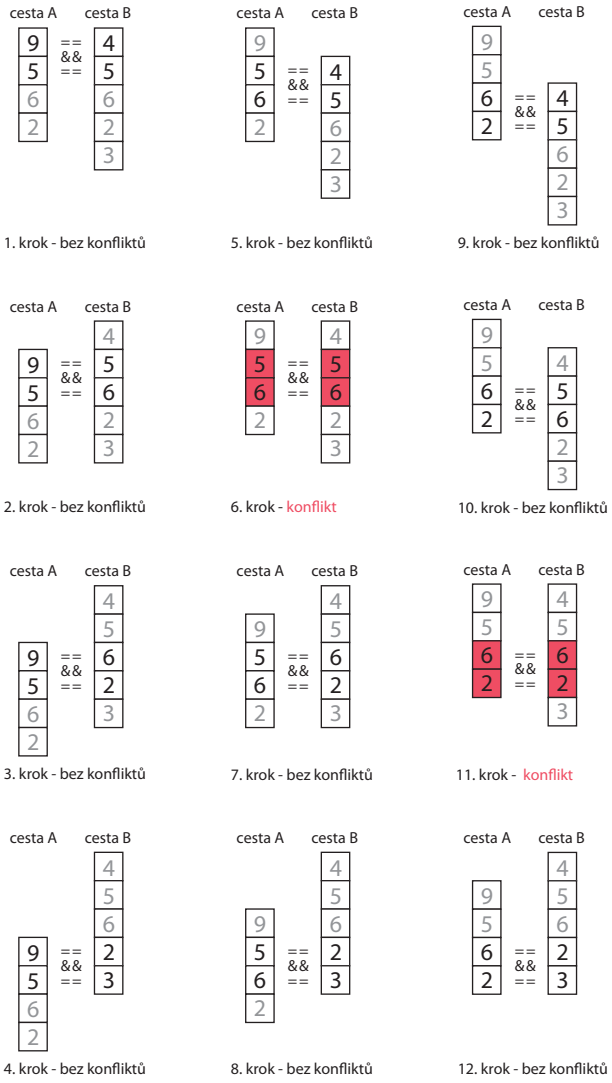
$$2(m_i - 1)(m_j - 1) \tag{6.2}$$

Celkový počet porovnání na jeden chromozóm v jednom časovém kroku potom vyjadřuje vzorec, který vznikne vynásobením dvou nadefinovaných vztahů 6.1 a 6.2:

$$\frac{n \cdot (n-1)}{2} \cdot 2(m_i - 1)(m_j - 1) \tag{6.3}$$

Pruběh výpočtu, který je prováděn současně každým vláknem lze shrnout v těchto bodech:

1. Získání hodnoty globálního ID vlákna.
2. Ukončení v případě, že ID vlákna je větší než počet cest naplánovaných na jeden krok.
3. Vynulování položky s indexem ID v poli pro výstup.
4. Porovnání cesty s indexem ID se všemi cestami s indexem větším než ID.
5. Uložení počtu nalezených konfliktů do výstupního pole na pozici s indexem ID.



Obrázek 6.3: Princip hledání konfliktů ve dvojici cest

6.2 Optimalizace výpočtu

Jelikož první verze aplikace zaměřená na dosažení funkčnosti neposkytuje žádné zrychlení oproti původní CPU implementaci, bylo nutné aplikovat řadu optimalizací. Tyto optimalizace zahrnují zejména efekektivnější práci s pamětí, zavedení nového způsobu porovnávání cest, větší využití GPU a paralelní redukci. Použité optimalizace jsou podrobně popsaný v následujících sekcích. Za účelem zjištění přínosu každé optimalizace, bylo vždy po její aplikaci provedeno testování. Tyto testy sloužily také pro ověření, že zavedením optimalizace nebyla do výpočtu zanesena chyba.

Cílem vytvářených optimalizací je upravit aplikaci tak, aby co nejlépe splňovala požadavky na dosažení vysokého výkonu definované Robem Farberem v [6]:

1. Minimalizace dat přenášených mezi GPU a operační pamětí počítače. Přenosy po sběrnici PCI snižují výkon aplikace.

2. Zajistit pro vlákna dostatek práce. Cílem je provést s daty na GPU maximum operací.

3. Provádět efektivní přistupy do paměti GPU.

6.2.1 Navýšení počtu vláken

Vyvíjená aplikace hledá pro zvolené topologie komunikační plány pro komunikace typu *all to all scatter* (viz 2.2). To znamená, že každý vysílající procesor odesílá zprávu každému přijímajícímu procesoru. V případě, že se všechny procesory chovají současně jako vysílače i přijímače, musí být provedeno právě p^2 komunikací, kde p je celkový počet procesorů v topologii.

Podle způsobu porovnávání popsaného v 6.1 je na GPU spouštěno tolik vláken, kolik je cest naplánovaných na jeden komunikační krok. I v případě reálných topologií obsahujicích desítky procesorů budou na GPU spouštěny maximálně stovky až jednotky tisíc vláken, což zdaleka nedosahuje potřebného vytížení současných GPU [15]. Pro dosažení efektivní práce na GPU je nutné vytížit všechna jádra, resp. multiprocesory na GPU. Specifikace GPU čipů na kterých bylo testování prováděno je uvedena v tabulce 6.3.

Dalším slabým místem algoritmu popsaného v 6.1 je kromě nedostatečného vytížení GPU samotný způsob porovnávání, který má za následek nerovnoměrné zatížení vláken. Délka výpočtu se pak odvíjí od doby běhu nejvíce zatíženého vlákna. Tabulka 6.1 znázorňuje způsob porovnávání a vysvětluje důvod nerovnoměrného zatížení vláken. Z této tabulky vyplývá, že první vlákno provádí $n-1$ porovnání, zatímco poslední vlákno pouze jediné porovnání.

Tabulka 6.1: Způsob přiřazení dvojic cest k porovnání jednotlivými vlákny

ID vlákna	rozpis porovnávaných dvojic cest								
0	$[0,1]$	$[0,2]$	$[0,3]$	$[0,4]$	$[0,5]$	$\dots$	$[0,n-2]$	$[0,n-1]$	$[0,n]$
1		$[1,2]$	$[1,3]$	$[1,4]$	$[1,5]$	$\dots$	$[1,n-2]$	$[1,n-1]$	$[1,n]$
2			$[2,3]$	$[2,4]$	$[2,5]$	$\dots$	$[2,n-2]$	$[2,n-1]$	$[2,n]$
3				$[3,4]$	$[3,5]$	$\dots$	$[3,n-2]$	$[3,n-1]$	$[3,n]$
4					$[4,5]$	$\dots$	$[4,n-2]$	$[4,n-1]$	$[4,n]$
$\vdots$		$\vdots$			$\vdots$			$\vdots$	$\vdots$
$n-3$							$[n-3,n-1]$	$[n-3,n-1]$	$[n-3,n]$
$n-2$								$[n-2,n-1]$	$[n-2,n]$
$n-1$									$[n-1,n]$

Oba zmíněné nedostatky původního návrhu byly odstraněny návrhem nového způsobu přiřazování cest k porovnání na jednotlivá vlákna. Cílem bylo vytvořit takový způsob přiřazování, který zabezpečí rovnoměrné vytížení vláken a dále zajistí navýšení celkového počtu vláken bez potřeby zavádění pomocných polí a bez použití podmíněných příkazů.

Jelikož je možné počet dvojic cest určených k porovnání stanovit ze vzorce 6.1 před samotným zahájením výpočtu na GPU, může být vytvořeno pravě takový počet vláken odpovídající počtu dvojic cest. Každé vlákno pak porovnává pravě jednu dvojici cest. Problémem realizace tohoto návrhu bylo přiřazení dvou cest vláknům tak, aby každé vlákno mělo unikátní dvojici cest a žadná dvě vlákna neprováděla shodná porovnávání. Jedinou informací, pomocí které je možné od sebe jednotlivá vlákna rozlišit na urovni kernelu, je

globální ID vlákna. Je-li kernel spuštěn s n vlákny, vrátí tato funkce hodnotu z intervalu $< 0, n-1 >$. Na základě pouze této hodnoty je nutné přiřadit dvě cesty tak, aby žadné jiné vlákno tuto kombinaci cest nezískalo. Máme-li n dvojic cest a vlákno s indexem ID, zajistí spolehlivé namapování následující dva vzorce:

$$\begin{array}{ll} \text{1. cesta:} & \text{ID mod } n \\ \text{2. cesta:} & (\text{ID} + \frac{\text{ID}}{n} + 1) \text{ mod } n \end{array} \tag{6.4}$$

Platnost těchto vzorců byla experimentálně ověřena. Následující úkázka slouží pro ilustraci namapování dvojic cest na jednotlivá vlákna. Uvažujme, že na jeden komunikační krok je naplánováno 6 komunikací, je tedy nutné porovnat 6 cest. Podle vzorce 6.1 tak vzniká 15 dvojic, jejichž namapování na vlákna podle vzorců 6.4 znázorňuje tabulka 6.2.

Tabulka 6.2: Optimalizovaný způsob přiřazování cest na vlákna

ID vlákna	Dvojice
0	[0, 1]
1	[1, 2]
2	[2, 3]
3	[3, 4]
4	[4, 5]
5	[5, 0]
6	[0, 2]
7	[1, 3]
8	[2, 3]
9	[3, 5]
10	[4, 0]
11	[5, 1]
12	[0, 3]
13	[1, 4]
14	[2, 5]

Každé vlákno si tedy na základě svého globálního ID vypočte indexy dvou cest, provede jejich porovnání a uloží počet nalezených konfliktů do výstupního pole. Aplikace této optimalizace zajistila rovnoměrnější vytížení vláken a vyšší vytížení GPU při zachování jednoduchosti výpočtu. Původní vzorec vyjadřující počet všech nutných operací porovnání lze rozdělit na část $\frac{n \cdot (n-1)}{2}$ vyjadřující počet spouštěných vláken a část $2(m_i - 1)(m_j - 1)$ určující počet porovnání, které musí každé vlákno vykonat.

6.2.2 Minimalizace prováděného kódu

Cílem této optimalizace bylo upravit funkce, které jsou volány v cyklech tak, aby obsahovaly minimum kódu. Zejména se jednalo o funkci `launchGPU()` popsanou v úvodu kapitoly 6. Tato funkce je volána v cyklu pro každý chromozom a každý časový krok. Uvnitř funkce probíhá časově náročná dynamická alokace v globálním paměťovém prostoru grafické karty pro vstupní pole s cestami naplánovanými na jeden časový krok a také pro výstupní pole s konflikty. Dynamická alokace značně snižovala rychlost provádění kódu, nemohla však být úplně odstraněna, jelikož potřebná délka vstupního a výstupního pole se pro každého jedince obecně liší a může být zjištěna až po jeho vygenerování.

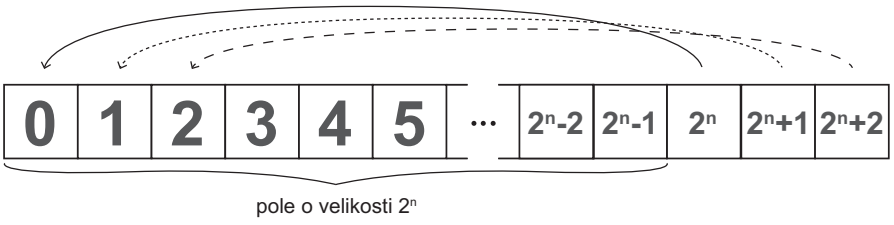
Na druhou stranu je možné již během inicializace výpočtu na základě parametrů zvolené topologie zjistit maximální možný počet cest naplánovaných na jeden časový krok. Díky tomu lze určit maximalní možnou velikost potřebnou pro vstupní i výstupní pole. Vstupní pole `sameStepPaths` obsahující cesty naplánované na jeden časový krok může obsahovat maximálně $3rt$ položek, kde r je počet vysílajících procesorů a t počet přijímajících procesorů v topologii. (viz 6.1) Položek ve výstupním poli pro konflikty pak může být maximálně rt. Ve většině případů budou tato pole naddimenzována, jejich alokaci lze však při znalosti maximálních možných délek provést pouze jedenkrát v rámci celého běhu aplikace a přesunout ji tak do funkce `initGPU()`. Tato optimalizace tedy docílila zrychlení výpočtu na úkor paměťové náročnosti.

6.2.3 Paralelní redukce

V původním návrhu aplikace byly konflikty nalezené každým vláknem po dokončení výpočtu na GPU uloženy do pole v globální paměti a překopírovány zpět do operační paměti počítače a následně v hostilské aplikaci sečteny. V případě počtu vláken odpovídajícímu počtu cest naplánovaných na jeden časový krok bylo položek určených k sečtení relativně málo. Maximálně se jich v tomto případě může vyskytnout $r \cdot t$ (viz 6.2.2). V topologiích s desítkami procesorů je tedy nůtné sečíst přibližně stovky nebo maximálně jednotky tisíc položek. Avšak v případě navýšeného počtu vláken, kdy je na GPU spouštěn počet vláken odpovídající počtu dvojic cest, dostáváme $\frac{rt \cdot (rt-1)}{2}$ položek, které je nutné na procesoru sekvenčně sečíst.

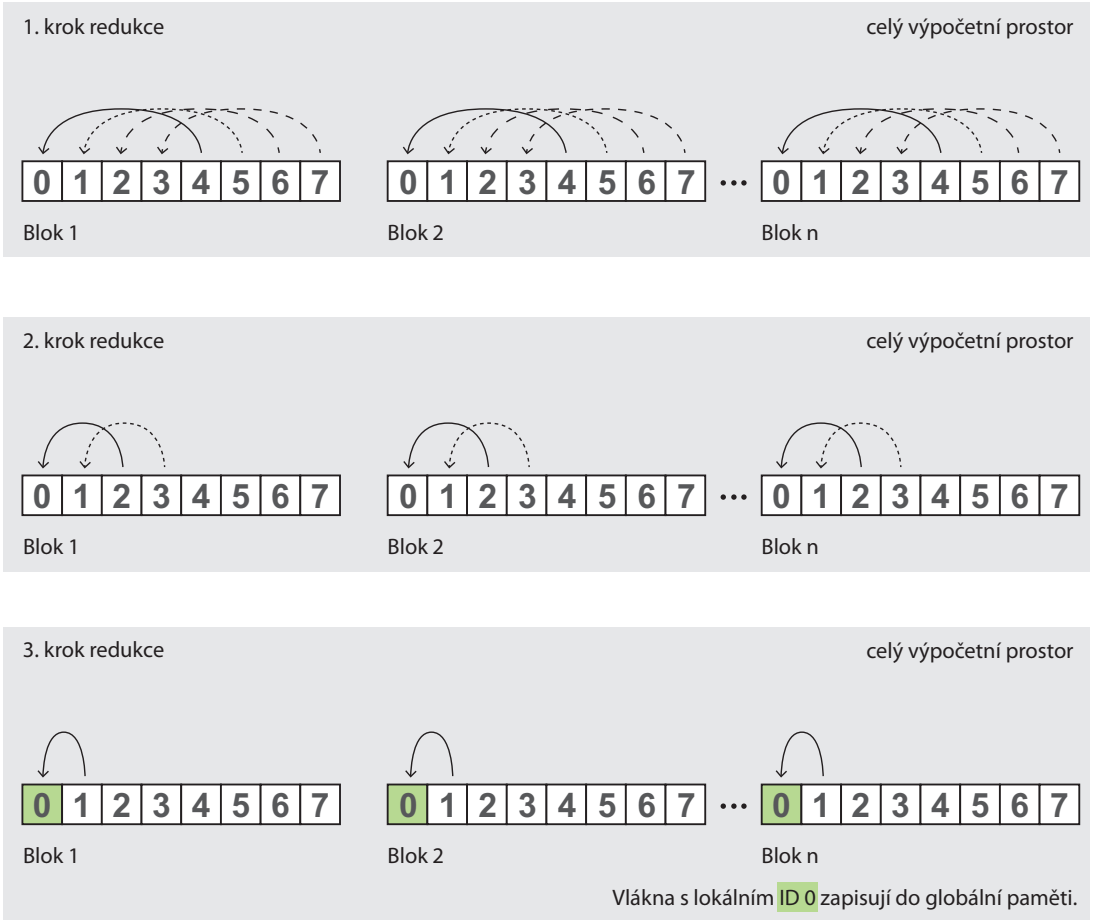
Kromě neefektivního způsobu sčítání je další slabinou tohoto řešení nutnost kopírovat velké množství dat z globální paměti grafické karty. Pro získání sumy konfliktů pro jeden chromozom v daném časovém kroku efektivnějším způsobem a bez nutnosti přesunu velkého množství položek mezi pamětí grafické karty a operační pamětí počítače byla zvolena paralelní redukce na straně GPU. Oproti sekvenčnímu sčítání na CPU, jehož složitost je $\mathcal{O}(n-1)$ se s paralelní redukcí dostáváme na logaritmickou složitost $\mathcal{O}(log_2 n)$. Je však nutné zajistit, aby byl počet redukovaných položek mocninnou dvou. Při každém kroku redukce je totiž sečtena polovina položek a řídící proměnná cyklu je vydělěna dvěma, není-li tedy počet redukovaných položek mocninnou dvou, dojde k předčasnému ukončení redukce a hodnoty nebudou korektně sečteny.

Prvním realizovaným způsobem vyřešení tohoto problému bylo sekvenční sečtení položek, které již nemohly být paralelně redukovány. Pokud však redukce skončila příliš brzy, bylo ve výsledku provedeno sekvenční sčítání na GPU, které je z důvodu taktovací frekvence CUDA jader (viz 6.3) pomalejší, než kdyby data byla přenesena a sečtena na CPU. Druhý způsob navrhuje před zahájením celkové redukce provést jeden krok pomocné redukce s takovým počtem položek, aby velikost výsledného pole určeného k hlavní redukci splňovala požadavek na mocninu dvou. Tento postup je schematicky znázorněn na obrázku 6.4.



Obrázek 6.4: Úprava velikosti pole na 2^n položek

Vzhledem k tomu, že je při redukci vždy polovina vláken nevyužita, je jako finální řešení zvolena redukce na úrovni bloků místo na úrovni celého kernelu. Výsledkem po dokončení výpočtu na grafické kartě je pole o velikosti počtu bloků. Na CPU poté proběhne sečtení jen několika položek, jejichž počet odpovídá počtu bloků. Nastavení velikost bloku neboli počtu vláken v jednom bloku je podrobněji diskutováno v 6.2.7. Vždy lze velikost bloku nastavit tak, aby byla mocninou dvojky. To umožňuje efektivně vyřešit problém s požadavkem na velikost redukovaných dat a současně se vyhneme sekvenčnímu sčítání velkého počtu položek a také redundantnímu kódu, což by si předešlé způsoby řešení vyžádaly. Obrázek 6.5 ilustruje způsob provádění paralelní redukce na úrovni bloků v rámci kernelu na GPU.



Obrázek 6.5: Paralelní redukce na úrovni bloků

6.2.4 Využití lokalní paměti

Základní verze porovnávacího algoritmu popsaného v 6.1, využívala téměř pro veškerou činnost globální paměť grafické karty. Jak je uvedeno v kapitole 4.2, jsou přístupy do této paměti časově náročné a dochází tak ke značnému poklesu výkonu aplikace. Z tohoto důvodu byla aplikace upravena tak, aby byla průběžná data získávaná během výpočtu ukládána do proměnných v privátním a lokálním paměťovém prostoru.

Před spuštěním kernelu je během inicializace ve funkci `initGPU()` v lokálním paměťovém prostoru grafické karty alokováno pole typu `int` s názvem `conflicts`. Toto pole je

vytvořeno v lokální paměti každého bloku, jejich počet tedy odpovídá počtu bloků spuštěných v rámci kernelu. Pole `conflicts` slouží pro uložení konfliktů nalezených každým vláknem během porovnávání cest a jeho velikost proto odpovídá počtu vláken, které tvoří jeden blok. Možnosti nastavení velikosti bloku, včetně maximální a doporučené velikosti jsou podrobněji popsány v části 6.2.7.

Pole `conflicts` je na začátku kernelu vynulováno. Tato operace již probíhá paralelně a každé vlákno nuluje jedinou položku pole. Jednotlivá vlákna v každém bloku jsou na položky lokálního pole namapována na základě lokálního ID, které je každému vláknu vráceno funkcí `get_local_id()`. Následně během porovnávání dvojice cest inkrementuje vlákno při nalezení konfliktu příslušnou položku pole `conflicts`. Finální podoba paralelní redukce probíhající na úrovni bloku, která je popsaná v části 6.2.3, také využívá výhod lokální paměti a pracuje přímo s polem `conflicts`. Jelikož je redukce prováděna v cyklu na úrovni lokální paměti, je po každém jejím kroku z důvodu zachování konzistence dat nutné explicitně využít synchronizační mechanismus. V kernelu jsou za tímto účelem použity lokální bariéry – `barrier(CLK_LOCAL_MEM_FENCE)`. Po dokončení redukce se na položce s indexem nula v poli `conflicts` nachází součet nalezených konfliktů v daném bloku. Vlákno s lokálním ID nula tuto hodnotu zapíše do výstupního pole v globální paměti na položku danou jeho globálním ID.

6.2.5 Načítání položek do privátní paměti

Jak bylo uvedeno v 6.1, je pro porovnání dvou cest nutné porovnat každý segment první cesty s každým segmentem druhé cesty. Této funkcionality je v kódu dosaženo pomocí dvou vnořených cyklů `for`. Jelikož je segment cesty jednoznačně určen dvěma uzly uloženými v globální paměti, je pro porovnání dvou segmentů zapotřebí čtyř přístupů do globální paměti. Narozdíl od CPU totiž výpočetní jádra GPU nemají vyrovnávací paměti. Ukázka kódu 6.1 představuje základní implementaci porovnání dvojice cest a sečtení nalezených konfliktů.

```
1           for (int l1 = 0; l1 < path1Size - 1; l1++)
2           {
3               for (int l2 = 0; l2 < path2Size - 1; l2++)
4               {
5                   if ((path1[l1] == path2[l2]) && (path1[l1 + 1] == path2[l2 + 1]))
6                       conflicts[iGID]++;
7               }
8           }
```

Kód 6.1: Porovnání jedné dvojice cest

Z principu porovnávání cest vyplývá, že jeden segment první cesty je porovnán se všemi segmenty druhé cesty. Z tohoto důvodu lze segment načíst do privátní paměti vlákna ve vnějším cyklu a načtené hodnoty využívat při porovnávání ve vnitřním cyklu. Privátní paměť je přístupná jen danému vláknu a lze na ni pohlížet jako na registry v procesoru. Podrobný popis hierarchie pamětí grafických karet včetně doby přístupu, rychlosti čtení a zápisu jsou uvedeny v kapitole 4.2. Pokud aplikujeme popsanou paměťovou optimalizaci a použijeme načítání segmentů do privátní paměti, klesne počet přístupů do globální paměti z $4(m_i - 1)(m_j - 1)$ na $2(m_i - 1) + 2(m_i - 1)(m_j - 1)$, což u přibližně stejně dlouhých cest představuje snížení zhruba o 40 %. m_i a m_j představují délky porovnávaných cest ve smyslu popsaném v 6.1.

Jelikož jsou při dvou po sobě následujících iteracích porovnávány sousední segmenty (viz obrázek 6.3), je každý uzel druhé cesty (kromě prvního a posledního) použit ve dvou ite-

racich vnitřního cyklu. Na základě této úvahy lze algoritmus dále optimalizovat a počet přístupů do globální paměti ještě snížit. Uzel, který byl použit pro porovnávání v jedné iteraci a má být využit v následující iteraci, předáme mezi privátními proměnnými vlákna namísto jeho opětovného načítání z globální paměti. Aplikací této optimalizace klesne počet přístupů na $1+2(m_i-1)+(m_i-1)(m_j-1)$, tedy přibližně o 70 % oproti původní neoptimalizované verzi algoritmu. Ukázka kódu 6.2 reprezentuje optimalizovanou verzi porovnávacího algoritmu.

```
1           node1_1 = path1[0]; // dopředné načtení 1.uzlu 1.cesty
2
3           for (int l1 = 0; l1 < path1Size - 1; l1++)
4           {
5               node1_2 = path1[l1 + 1];
6               node2_1 = path2[0];     // dopředné načtení 1.uzlu 2.cesty
7
8               for (int l2 = 0; l2 < path2Size - 1; l2++)
9               {
10                  node2_2 = path2[l2 + 1];
11
12                  // porovnání segmentu a případná inkrementace počtu konfliktů
13                  if ((node1_1 == node2_1) && (node1_2 == node2_2))
14                      conflicts[iGID]++;
15
16                  node2_1 = node2_2; // 2.uzel 2.cesty je uschován pro další iteraci
17              }
18              node1_1 = node1_2; // 2.uzel 1.cesty je uschován pro další iteraci
19          }
```

Kód 6.2: Optimalizované porovnání dvojice cest

6.2.6 Využití více dimenzí

Pro dosažení maximalního výkonu jakékoliv aplikace prováděné na GPU je jedním ze stěžejních bodů zajištění co nejvyššího vytížení výpočetních jader. Pokud je výpočet na GPU spouštěn opakovaně a pro malé problémy, bude mít režie vznikající kopírováním dat a spouštěním kernelů negativní vliv na celkový výkon aplikace. V původní verzi aplikace byl výpočet spouštěn vždy pro každý chromozom a každý časový krok. Počet spuštění kernelu a přenosů po PCI sběrnici mezi globální pamětí zařízení a operační pamětí počítače v rámci jedné generace potom odpovídá součinu velikosti populace a počtu časových kroků.

Za účelem snížení režie, vznikající častým spouštěním kernelu a vysokým počtem paměťových přenosů, byla aplikace optimalizována pomocí využití vícedimenzionálních bloků. OpenCL umožňuje využít až třídimenzionální bloky vláken. Maximální velikost v každé dimenzi je závislá na konkrétním zařízení a lze ji zjistit během výpočtu voláním funkce `clGetDeviceInfo()` s parametrem `CL_DEVICE_MAX_WORK_ITEM_SIZES`. Informace o počtu dimenzí a jejich požadované velikosti se předávají funkci `clEnqueueNDRangeKernel()`, která spouští kernel. Zde je třeba rozlišit pojmy *lokální* a *globální pracovní velikost* (anglicky *local work size* a *global work size*).

- **Global work size** určuje velikost každé dimenze na globální úrovni kernelu. Vyjadřuje požadavek na celkový počet vláken v rámci provádění kernelu. Velikosti dimenzí se specifikují polem o takovém počtu položek jako je požadovaný počet dimenzí. Celkový počet vláken daného kernelu je pak možné určit jako součin velikostí ve všech dimenzích: `globalWorkSize[0] * globalWorkSize[1] * globalWorkSize[2]`

- **Local work size** určuje velikost každé dimenze bloku. Vyjadřuje požadavek na počet vláken tvořících jeden blok. Velikosti dimenzí se specifikují polem o takovém počtu položek jako je požadovaný počet dimenzí. Celkový počet vláken tvořící jeden blok daného kernelu je pak možné určit jako součin velikostí ve všech dimenzích: `localWorkSize[0] * localWorkSize[1] * localWorkSize[2]` Celkový počet bloků spuštěných během provádění kernelu potom vyjadřuje součin podílů odpovídajících si dimenzí globálních a lokálních pracovních velikostí:

$$\text{počet bloků} = \frac{\texttt{globalWorkSize[0]}}{\texttt{localWorkSize[0]}} \cdot \frac{\texttt{globalWorkSize[1]}}{\texttt{localWorkSize[1]}} \cdot \frac{\texttt{globalWorkSize[2]}}{\texttt{localWorkSize[2]}}$$

Kromě požadavku na dodržení maximálního rozměru každé dimenze platí, že nesmí být překročena maximální velikost bloku. Tato hodnota je opět specifická pro každé zařízení a může být zjištěna voláním funkce `clGetKernelWorkGroupInfo()` s parametrem `CL_KERNEL_WORK_GROUP_SIZE`. Další důležitou podmínkou při specifikaci pracovních velikostí je dělitelnost beze zbytku u všech odpovídajících si dvojic globálních a lokálních pracovních velikostí. Při nedodržení této podmínky vrací funkce `clEnqueueNDRangeKernel()` chybu `CL_INVALID_WORK_GROUP_SIZE`. Tato podmínka je ve většině aplikací, kde je explicitně nastavena lokální i globální pracovní velikost, zaručena zaokrouhlením globální pracovní velikosti na násobek lokální pracovní velikosti.

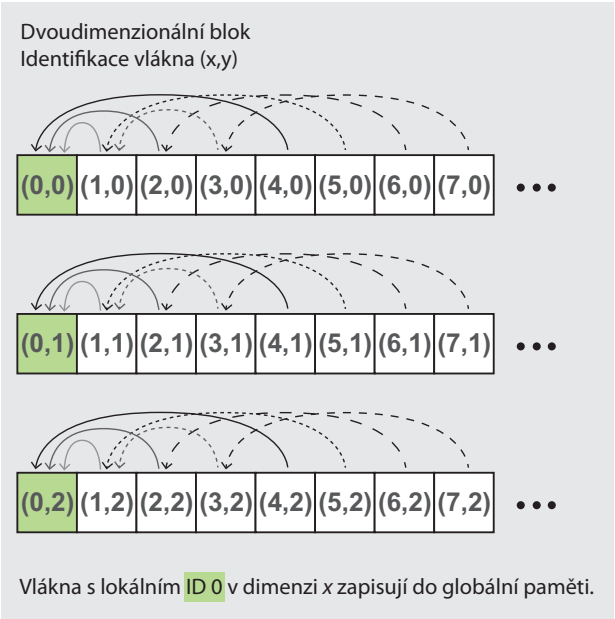
V první fázi byla přidána jedna dimenze, jejíž velikost odpovídala počtu časových kroků. Cílem bylo nalézt všechny konflikty jednoho chromozomu ve všech časových krocích při jednom spuštění kernelu. Počet spuštění kernelu pro ohodnocení jedné generace tak klesl na hodnotu odpovídající počtu chromozómů v populaci. Z implementačního hlediska si zavedení druhé dimenze vynutilo několik změn:

- **Indexováním vláken** – v případě jedné dimenze bylo možné každé vlákno jednoznačně identifikovat na základě jeho jediného globálního ID na úrovni kernelu a pomocí jediného lokálního ID v rámci bloku. Po zavedení druhé dimenze je vlákno identifikováno globálním a lokálním ID každé dimenze zvlášť. Z tohoto důvodu byly v privátní paměti vláken vytvořeny tyto proměnné:
  - `iGIDx` – globlní ID vlákna v první (původní) dimenzi, určuje dvojici cest
  - `iGIDy` – globlní ID vlákna ve druhé dimenzi, určuje časový krok
  - `iLIDx` – lokální ID vlákna v první (původní) dimenzi, určuje dvojici cest v bloku
  - `iLIDy` – lokální ID vlákna ve druhé dimenzi, určuje časový krok v rámci bloku

  Globální resp. lokální ID pro každou dimenzi vlákno získá na začátku kernelu zavoláním funkce `get_global_ID(x)` resp. `get_local_ID(x)`, kde `x` je obecně číslo dimenze, v tomto případě 0 nebo 1. Pro práci s vícedimenzionálními bloky vláken je zapotřebí znát velikost bloku v každé dimenzi, k tomu slouží funkce `get_global_size(x)` resp. `get_local_size(x)`. Například indexování lokálního pole pro počítání konfliktů se změnilo z `conflicts[iLIDx]` na `conflicts[iLIDy * get_local_size(0) + iLIDx]`.

- **Paralelní redukce** je prováděna vlákny na úrovni bloku. Při rozšíření bloku na dvě dimenze vznikly dvě možnosti provedení redukce. První možností bylo rozšířit redukci na obě dimenze tak, aby po jejím dokončení vznikla opět jediná hodnota vyjadřující počet nalezených konfliktů v celém bloku. Druhá možnost představuje zachovat redukci v původním stavu, tedy na úrovni jedné dimenze. Výsledkem redukce

by potom bylo pole sečtených konfliktů s počtem položek odpovídajícím hodnotě `get_local_size(1)`. Během fáze testování byly tyto možnosti prověřeny a za výsledný způsob provedení byla zvolena druhá popsaná možnost. Schéma znázorňující paralelní redukci na úrovni jedné dimenze ve dvoudimenzionálním bloku reprezentuje obrázek 6.6



Obrázek 6.6: Paralelní redukce na úrovni dvoudimenzionálního bloku

- **Velikost bloku** se zavedením nové dimenze zvýšila o násobek daný hodnotou `get_local_size(1)`. Vhodné nastavení lokálních pracovních velikostí v jednotlivých dimenzích je podrobněji popsáno v sekci 6.2.7. Větší velikost bloku si výžadala navýšení paměti alokované pro lokální pole `conflicts` a vzhledem k volbě provedení paralelní redukce musela být upravena také velikost výstupního pole.

Po odladění a otestování aplikace pracující s dvoudimenzionálními bloky byla přidána i třetí dimenze. Smyslem této dimenze je zahrnout celou populaci jedinců do výpočtu na GPU současně. Celá populace ve všech časových krocích je tak ohodnocena na jedno spuštění kernelu. Globální pracovní velikost v této dimenzi byla nastavena na počet jedinců v populaci. Lokalní pracovní velikost byla v tomto případě nastavena na hodnotu 1. Hodnota vyšší než jedna by způsobila znásobení velikosti bloku, což by mohlo způsobit nedostatek lokální paměti nebo překročení maximální velikosti bloku. Velikost bloku je již dostatečně naplněna hodnotami z prvních dvou dimenzí. Tento způsob také zabezpečil, že na jednom bloku nebudou ohodnocováni dva různí jedinci. Přidání třetí dimenze popsaným způsobem si tak nevyžádalo další navyšování velikosti lokálního pole `conflicts` ani další zasádnější úpravy kernelu. Větší úpravy musely být provedeny v hostitelské aplikaci. Jedinci v populaci jsou totiž implementováni jako objekty třídy `Chromozome` a jejich ohodnocení probíhalo voláním metody `Fitness()` každého chromozómu zvlášť. Tato skutečnost zabraňovala ohodnocení více jedinců současně. Z tohoto důvodu byla metoda `Fitness()` ve třídě `Chromozome` zrušena a obdobná metoda byla vytvořena na úrovni celé populace ve třídě `Population`.

V průběhu testování bylo zjištěno, že u rozsáhlejších topologií a velkých populací jedninců může spuštění kernelu skončit s chybou `CL_OUT_OF_RESOURCES`. Tuto chybu zbůsobuje vyčerpání zdrojů výpočetního zařízení a nejčastěji k ní dochází z důvodu nedostatku paměti. Po zavedení třetí dimenze jsou totiž do paměti zařízení zkopírována data pro celou populaci. Rozdíl v paměťové naročnosti oproti verzi se dvěma dimenzemi je vyjadřen násobkem velikosti ohodnocované populace. Do souboru `GPU.h` proto byla přidána konstanta `MAX_CHROMS_PER_CYCLE`, která vyjadřuje počet současně ohodnocovaných jedinců. Pokud je její hodnota větší než velikost populace, jsou všichni jedninci ohodnoceni současně. V opačném případě je populace rozdělena na částí podle hodnoty `MAX_CHROMS_PER_CYCLE` a tyto části jsou postupně zpracovány na GPU. Počet spuštění kernelu pro ohodnocení jedné generace udává hodnota c vyjádřená vzorcem 8.1.

$$c = \left\lceil \frac{\texttt{populationSize}}{\texttt{MAX\_CHROMS\_PER\_CYCLE}} \right\rceil \tag{6.5}$$

6.2.7 Přizpůsobení velikosti problému konkrétnímu zařízení

Při tvorbě aplikace bylo nutné předpokládat, že bude spouštěna na různých typech zařízení a na různých modelech grafických karet. Každá grafická karta nebo zařízení podporující OpenCL se však liší svými parametry. Z tohoto důvodu by při staticky nastavených hodnotách u paramatrů, udávajících například lokální pracovní velikost v jednotlivých dimenzich, nemohlo být dosaženo maximálního výkonu pro každé zařízeni. Pokud má aplikace optimálně fungovat na každém zařízení, je nutné přizpůsobit parametry výpočtu tak, aby odpovídaly specifikaci konkrétního zařízení.

Standard OpenCL v tomto ohledu nabízí několik funkcí, pomocí kterých lze před spuštěním kernelu určit parametry zařízení, na kterém bude výpočet probíhat. V aplikaci jsou za tímto účelem využity tyto funkce:

- `clGetDeviceInfo()` s parametrem `CL_DEVICE_MAX_WORK_ITEM_SIZES`
  - Volání funkce s tímto parametrem vrací maximalní velikosti bloku v jednotlivých dimenzích. V případě překročení velikosti některé z dimenzí, je při spuštění kernelu vrácena chyba `CL_INVALID_WORK_ITEM_SIZE`.
- `clGetKernelWorkGroupInfo()` s parametrem `CL_KERNEL_WORK_GROUP_SIZE`
  - Hodnota vrácená touto funkcí udává maximalní velikost bloku. Při překročení této hodnoty skončí spuštění kernelu s chybou `CL_INVALID_WORK_GROUP_SIZE`.
- `clGetKernelWorkGroupInfo()`
  s parametrem `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`
  - Při snaze dosáhnout maxímální výkonnosti aplikace na daném zařízení nestačí zaručit pouze jeho maximální vytížení. Velikost pracovní skupiny by měla být násobkem hodnoty, která je vrácena touto funkcí. Typicky se jedná o hodnotu odpovídající velikosti warpu.

Aplikace na základě těchto funkcí provede nastavení hodnot lokálních pracovních velikostí automaticky. Zároveň však byla zachována možnost tyto hodnoty nastavit ručně na požadovanou velikost. K tomu slouží konstanty `LOCAL_WORK_SIZE_X` a `LOCAL_WORK_SIZE_Y`

definované v souboru `GPU.cpp`. Jejich výchozí hodnota je nastavena na nulu, v tomto případě proběhne automatické nastavení. Hodnoty těchto konstant jsou použity pokud jsou explicitně nastaveny na hodnotu větší než nula. V průběhu inicializace jsou nastavené hodnoty otestovány a v případě, že jejich součin přesáhne hodnotu `CL_KERNEL_WORK_GROUP_SIZE` jsou tyto konstanty ignorovány a opět proběhne automatické nastavení. K tomuto dojde také pokud hodnota `LOCAL_WORK_SIZE_X` není mocninou dvojky, tento požadavek vznikl vzhledem ke způsobu provedení paralelní redukce popsané v 6.2.3.

Tabulka 6.3 uvádí parametry grafických karet, které byly použity během vývoje aplikace a na nichž bylo prováděno testování.

Tabulka 6.3: Parametry testovaných grafických karet

ID vlákna	Quadro FX 770M	GeForce GTX 285	GeForce GTX 580
Počet jader	32	240	512
Frekvence jader	1 250 MHz	1 505 MHz	1 564 MHz
Globální paměť	512 MB	2 048 MB	1 536 MB
Lokální paměť	16 KB	16 KB	48 KB
Max. velikost bloku	512	512	1024
Max. velikosti dim.	512/512/64	512/512/64	1 024/1 024/64
Registrů na blok	8 192	16 384	32 768
Počet multiprocesorů	4	30	16
Velikost warpu	32	32	32

6.2.8 Návrh použití bitových operátorů při hledání konfliktů

Tato optimalizace byla navržena za účelem akcelerace algoritmu provádějícího porovnání dvojice cest. Ukázka tohoto algoritmu je uvedena v sekci 6.2. Cílem optimalizace bylo z tohoto algoritmu eliminovat podmíněný příkaz `if`. Podmíněné příkazy na architekturách *SIMD* způsobůjí divergenci vláken, což má za následek zpomalení celého výpočtu [30].

Podmíněný příkaz `if`, při jehož splnění je inkrementován počet nalezených konfliktů:

```
if (node1_1 == node2_1 && node1_2 == node2_2)
    conflicts[iGID]++;
```

byl nahrazen následujícím kódem využívajícím bitové operátory:

```
conflicts[iGID] += !((bool) (node1_1 ^ node2_1 | node1_2 ^ node2_2));
```

Mezi porovnávanými dvojicemi segmentů je proveden `xor`. V případě, že jsou segmenty shodné, je výsledkem této operace nula. Tato hodnota je následně přetypována na datový typ `bool`, čímž vzniká hodnota `false`. Dále je provedena negace, jejímž výsledkem je `true` reprezentované číselnou hodnotou 1, která je následně přičtena k položce v poli `conflicts`. Pokud se segmenty neshodují vzniká po provedení logických operací `xor` a `or` nenulová hodnota, která je přetypováním a negací převedena na nulu. Testování sice potvrdilo, že výsledky jsou z hlediska správnosti v pořádku avšak výkonnostní měření ukázala, že rychlost aplikace klesla v řádu jednotek procent. Toto může být způsobeno velkým počtem logických operátorů, kterými byl jediný příkaz `if` nahrazen. Dalším úzkým místem snižujícím rychlost běhu může být použité přetypování pomocí `(bool)`. Z těchto důvodů nebyla tato navržená optimalizace do finální verze aplikace zahrnuta.

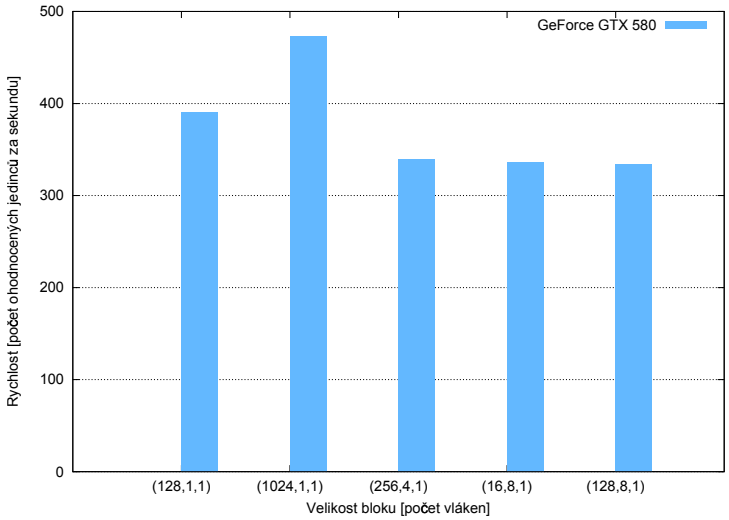
Kapitola 7

Interpretace dosažených výsledků

Tato kapitola prezentuje výsledky práce získané během vývoje a testování aplikace. Následující tabulky a grafy uvádí srovnání rychlosti běhu aplikace na několika testovaných zařízení. Specifikace grafických karet, na kterých testování probíhalo, je uvedena v tabulce 6.3. Testování dále probíhalo na dvoujádrovém procesoru Intel Core 2 Duo T9800 s frekvencí jádra 2.8 GHz a na čtyřjádrovém procesoru Intel Core i7 920 s frekvencí jádra 2.67 GHz. Pokud není uvedeno jinak, je při použití zkratky GPU myšlen čip GeForce GTX 580, za primární CPU je uvožován procesor Intel Core i7 920.

7.1 Nalezení optimální velikosti bloku

Zvolení vhodné velikosti bloku, tedy počtu vláken v pracovní skupině, má velký vliv na rychlost výpočtu. Tato velikost může být nastavena ručně pomocí konstant `LOCAL_WORK_SIZE_X` a `LOCAL_WORK_SIZE_Y` definovaných v souboru `GPU.cpp` nebo ji lze ponechat na automatickém nastavení způsobem popsaným v části 6.2.7. Výsledky testů pro grafickou kartu GeForce GTX 580 jsou uvedeny v grafu 7.1. Výsledky pro další testované grafické karty se nacházejí v příloze C. Parametry testu byly pro všechny grafické karty shodné. Testování proběhlo na topologii mřížka 7x7 s velikostí populace 200.



Obrázek 7.1: Rychlost ohodnocování v závislosti na velikosti bloku pro GeForce GTX 580.

Provedené testy ukázaly, že i přes zavedení dvoudimenzionálních bloků je u všech testovaných grafických karet nejvýhodnější nastavit velikost bloku v dimenzi y, která určuje počet řešených časových kroků současně v jednom bloku, na hodnotu jedna. Při nastavení velikosti bloku v dimenzi y na hodnotu větší než jedna dochází k patrné ztrátě výkonnosti. V tomto případě totiž po provedení paralelní redukce vzniká tolik hodnot, jako je velikost dimenze y. Výsledek do globální paměti tak zapisuje více vláken z jednoho bloku, případně také více vláken z jednoho warpu, což představuje nezarovnaný přístup do paměti (tzv. *non-coalesced memory access* [20]), jenž má negativní vliv na rychlost zpracování.

Nejlepších výsledků s grafickou kartou GeForce GTX 580 bylo dosaženo při nastavení velikosti bloku v dimenzi y na hodnotu jedna a v dimenzi x na hodnotu 1024, což odpovídá maximální velikosti v této dimenzi a celkově maximální velikosti bloku.

7.2 Dosažené zrychlení podle topologií

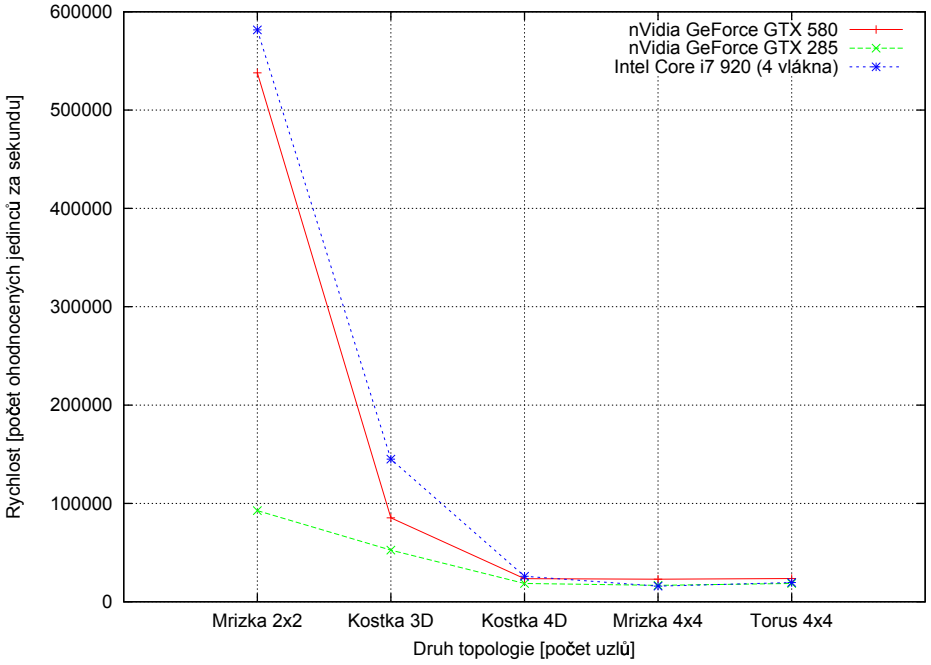
Již během vývoje aplikace bylo jisté, že velikost topologie bude mít zásádní vliv na úroveň zrychlení dosaženou pomocí GPU. Tato skutečnost byla potvrzena při testování na různých typech topologií. S rostoucí velikostí topologie výrazně klesá rychlost ohodonocování jedinců napříč všemi zařízeními. Uvažujeme-li například topologii se 16 procesory je nutné provést 256 komunikací, zvýšíme-li počet procesorů čtyřikrát na 64, bude nutné realizovat 4096 komunikací. Počet komunikací se tedy zvyšuje s druhou mocninou vzhledem k nárůstu počtu procesorů v topologii. Pokles rychlostí byl proto očekáván. Míra poklesu se však na různých zařízeních značně liší. Z tabulky 7.1 vyplývá, že úroveň poklesu rychlosti na procesoru Core i7 920 má výrazněji strmější charakter než na grafické kartě GeForce GTX 580.

Tabulka 7.1: Počet ohodnocených jedinců za sekundu v různých topologiích

Topologie	GeForce GTX 285	GeForce GTX 580	Core i7 920 (4 vlákna)
Mřížka 2x2	92 692	537 895	581 530
Mřížka 3x3	48 369	113 188	89 572
Mřížka 4x4	16 741	22 981	15 930
Mřížka 5x5	3 810	7 563	3 588
Mřížka 6x6	750	2 372	504
Mřížka 7x7	108	539	112
Kostka 3D	52 454	85 384	14 5077
Kostka 4D	18 621	23 456	25 845
Kostka 5D	1 835	4 152	1 369
Kostka 6D	51	222	65
Torus 4x4	18 619	23 551	19 498
Torus 4x5	11 835	14 236	8 768
Torus 5x5	6 336	8 353	3 695
Torus 5x6	2 898	4 979	1 625
Torus 4x8	1 774	3 946	1 119
Torus 6x6	630	2 709	677
Torus 7x7	351	725	176

Tyto testy potvrdily, že zrychlení dosažené zpracováním na GPU se zvyšuje s počtem

procesorů v topologii. Maximálního zrychlení bylo dosaženo na topologii typu mřížka se 49 procesory, zrychlení dosáhlo hodnoty 4.8125. Naměřená data z tabulky 7.1 byla rozdělena do dvou skupin (na menší a větší topologie) a vynesena do grafů. Srovnání rychlostí ohodnocování na meších topologiích s počtem procesorů menším než dvacet je uvedeno v grafu 7.2. Z tohoto grafu vyplývá, že výpočet na CPU je rychlejší, než na obou testovaných GPU. Nižší rychlost GPU je způsobena režií potřebnou pro inicializaci kernelu a přenos dat do globální paměti grafické karty. Samotný výpočet je pak u takto malých topologií příliš krátký na to, aby plně vytížil GPU a překryl režii.



Obrázek 7.2: Porovnání rychlosti ohodnocování na menších topologiích. Velikost populace nastavena na 200 jedinců.

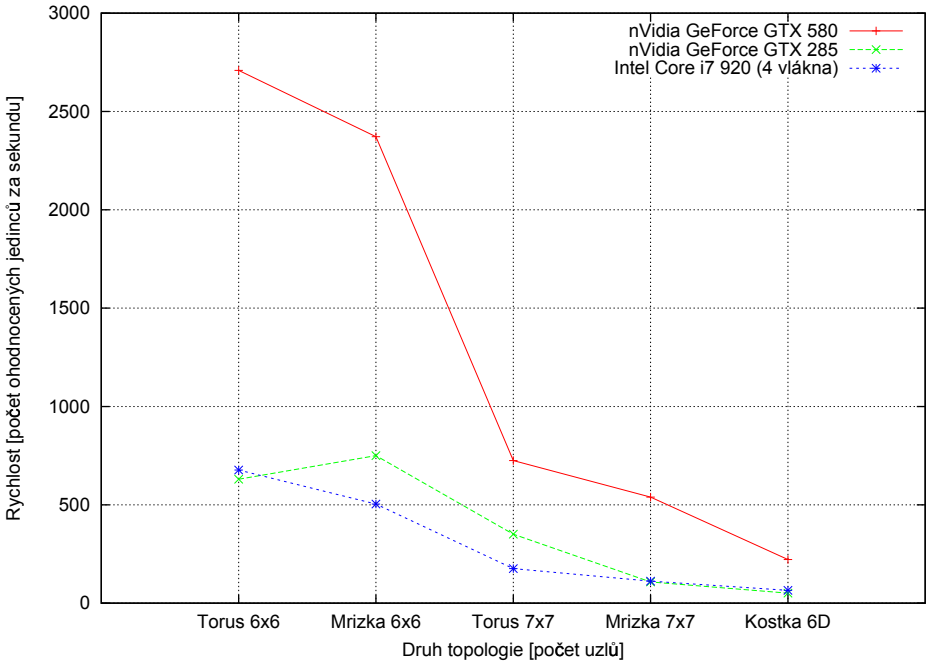
Výsledky porovnání rychlosti u větších topologií jsou vyneseny v grafu 7.3. V tomto případě již režie tvoří pouze zlomek výpočetní doby a zrychlení oproti zpracování na CPU narůstá. Toto je důležité zjištění, jelikož dosáhnout zrychlení na malých topologiích, které je možné řešit v řádu minut, není prakticky významné. V rámci disertační práce doktora Jaroše byla na CPU řešena například topologie 5D kostka s 32 procesory za čtyři dny a pět hodin. Při použití akcelerovaného výpočtu na GPU by byl bezkolizní plán kolektivních komunikací typu *all to all scatter* pro tuto topologii nalezen za jeden den a devět hodin

Zajimavým zjištěním během fáze testování byl značný rozdíl v rychlosti ohodnocování mezi grafickými kartami GeForce GTX 285 a GeForce GTX 580. Rozdíl v rychlosti porovnání byl téměř pětinásobný avšak výkonnostní rozdíl mezi těmito kartami je přibližně jen 50 %[1]. Srovnání těchto karet je uvedeno také v tabulce 6.3. nVidia rozděluje tyto karty do dvou různých skupin [2] z hlediska výpočetních možností (tzv. *Compute capability*)[3].

Hlavní předností GPU GeForce GTX 580, který se označuje kódovým názvem *fermi* [27],

[1] http://www.gpureview.com/show_cards.php?card1=605&card2=637
[2] http://developer.nvidia.com/cuda-gpus
[3] http://blog.cuvilib.com/2010/06/09/nvidia-cuda-difference-between-fermi-and-previous-architectures/

je přítomnost vyrovnávací paměti umožňující rychlé uspořádání dat za účelem provedení zaravonaného přístupu do paměti. Dvojnásobný počet výpočetních jader u fermi umožňuje současný běh dvojnásobného počtu vláken, čímž dochází k lepšímu vykrývání režie při přístupu do globální paměti.



Obrázek 7.3: Porovnání rychlosti ohodnocování na menších topologiích. Velikost populace nastavena na 200 jedinců.

Kapitola 8

Rozšíření na více platforem

Původním záměrem práce bylo docílit akcelerace výpočtů výhradně pomocí jednotky GPU. Mezi výpočetní platformy podporované standardem OpenCL[1] však kromě grafických karet patří také procesory společností AMD a Intel a dále Blade systémy společnosti IBM[2]. Z tohoto důvodu bylo navrženo rozšíření práce, které by umožnilo provádět paralelizovanou část aplikace (algoritmus hledání konfliktů) na všech podporovaných platformách. Zpracování OpenCL kódu na běžných procesorech vyžaduje speciální ovladače, z tohoto důvodu byl otestován pouze multi GPU režim. Výsledky jsou prezentovány v části 8.2.

8.1 Popis implementace multiplatformního rozšíření

Realizace tohoto rozšíření byla provedena v závěru implementační fáze. Bylo nutné provést zásadní změny ve funkci `initGPU()`, jejíž původní účel – alokovat prostředky na grafické kartě a inicializovat výpočet na jediném GPU čipu – bylo nutné upravit pro obecný typ i počet zařízení. OpenCL API vytváří abstraktní vrstvu nad heterogenními vypočetními zařízení a poskytuje funkce umožňující jednotnou práci s těmito zařízeními. V následujícícm textu je uveden postup při implementaci tohoto rozšíření.

Pro získání seznamu všech výpočetních platforem dostupných v systému je využita funkce `clGetPlatformIDs()`. Pro každou nalezenou platformu je následně volána funkce `clGetDeviceIDs()`, která vrací pole ukazatelů na dostupná zařízení. Na standardní výstup je poté vypsán seznam jmen nalezených zařízení, který je získán funkcí `clGetDeviceInfo()`. Tento postup byl zvolen s cílem, dát uživateli možnost volby, na kterých zařízeních bude výpočet probíhat. Interakce s uživatelem probíhá v konzoli jednoduchým dotazováním a následně načtením zvolené možnosti přes standardní vstup. Aby mohla být aplikace spouštěna jako součást skriptů byla implementována možnost vypnutí interakce s uživatelem pomocí konstanty `GET_ALL_DEVS`. Implicitní hodnota této konstatny je nastavena na `false`. Při nastavení na hodnotu `true` je možnost výběru zařízení přeskočena a jsou automaticky vybrána všechna dostupná zařízení.

Kód ve funkci `launchGPU()`, zahrnující zápis cest naplánovaných na jeden časový krok do paměti grafické karty, spuštění kernelu a čtení výstupního pole s konflikty, je nyní prováděn v cyklu pro každé zvolené zařízení. Přičemž operace čtení a zápisu dat jsou provedeny asynchronně. Jelikož se nečeká na dokončení operací, je celý popsaný cyklus i v případě většího počtu zařízení proveden velmi rychle. Všechny tyto operace totiž spočívají v uložení

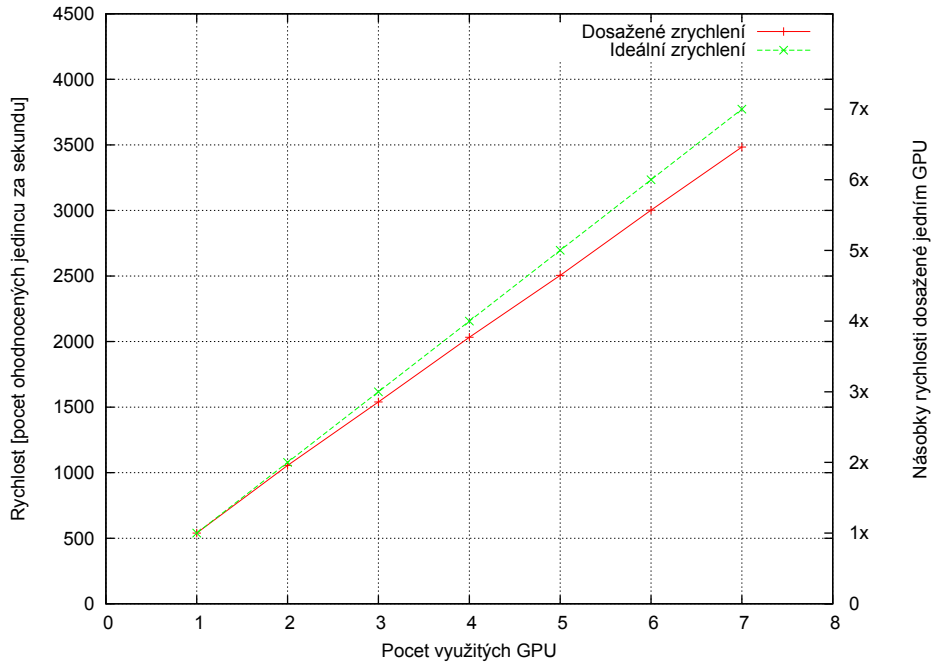
[1] http://www.khronos.org/conformance/adopters/conformant-products/
[2] http://www-03.ibm.com/systems/cz/bladecenter/

příkazu do příkazové fronty (tzv. *command queue*) daného zařízení. Operace z těchto front jsou prováděny příslušnými zařízeními bez ohledu na stav volajícího procesu. Synchronizace probíhá až před zápisem nalezených konfliktů do proměnné fitness jednotlivých chromozomů. Synchronizace je realizována pomocí funkce `clWaitForEvents()`.

8.2 Multi GPU režim

Implementované rozšíření bylo otestováno na serveru Australské národní univerzity[3], který je vybaven sedmi grafickými kartami nVidia GeForce GTX 580 [12]. Cílem testu bylo zjistit dosažené zrychlení při distribuci populace určené k ohodnocení mezi více stejně výkonných zařízení. Způsob distribuce výpočtu mezi jednotlivá zařízení je popsán v části 8.3. Graf 8.1 vyjadřuje počet ohodnocených jedinců za jednotku času v závislosti na počtu pracujících grafických karet.

Základní velikost populace byla při tomto testu nastavena na 256. Celková velikost populace, se kterou testování probíhalo je dána součinem základní velikosti populace a počtem GPU zapojených do výpočtu. Ideálního zrychlení, které je v grafu vyneseno zeleně a reprezentuje násobky základní rychlosti při použití jednoho GPU, nemůže být dosaženo, jelikož distribuce jedinců mezi několik zařízení a následné čtení výsledků představují nutnou režii.



Obrázek 8.1: Dosažené zrychlení při distribuci jedinců na 1 až 7 grafických karet

[3] http://www.anu.edu.au/

8.3 Distribuce zátěže

Distribuce výpočtu mezi jednotlivá zařízení je prováděna na úrovni jedinců. Jedinci, kteří mají být ohodnoceni jsou rovnoměrně rozděleni mezi všechna zařízení podle následujícího způsobu:

$$\texttt{chromozomsPerDevice} = \left\lceil \frac{\texttt{populationSize}}{\texttt{deviceSum}} \right\rceil \tag{8.1}$$

Tento způsob rozdělení jedinců není ideální, protože neuvažuje obecně vysokou rozdílnost ve výkonnosti u výpočtních zařízeních dostupných v systému. V případě aplikace bez nutnosti synchronizace by rozdílný výpočetní výkon nepředstavoval problém, v evolučních algoritmech je však nutné synchronizaci provést mezi každou generací. Při rovnoměrném rozdělení jedinců je tak výpočet zpomalován méně výkonnými zařízeními.

Při testování se tento problém neprojevil, protože testy probíhaly na stejně výkonných grafických kartách (viz 8.2). Pro reálné využití aplikace by musel být způsob distribuce jedinců upraven tak, aby zatížení jednotlivých zařízení odpovídalo jejich výkonu. Možným řešením je provedení zátěžového testu všech zařízení během fáze inicializace a následná distribuce jedinců na základě výsledků těchto testů. Podle naměřených výsledků uvedených v tabulce 7.1 je výpočet na grafické kartě GeForce GTX 580 na některých topologiích téměř 5x rychlejší než čtyřvláknové zpracování na procesoru Core i7 920. Pokud by tedy tyto dvě zařízení měli pracovat současně bylo by pro dosažení maximální výkonnosti nutné nastavit poměr distribuce jedinců mezi grafickou kartu a procesor přibližně na 5:1.

8.4 Zhodnocení multiplatformního řešení

Z pohledu hardwaru toto rozšíření značně zvýšilo přenositelnost aplikace napříč širokým spektrem výpočetních zařízení podporovaných OpenCL. Toto rozšíření také naznačuje možný budoucí směr vývoje aplikace, tedy využití všech dostupných výpočetních zařízení daného systému namísto pouze nejvýkonějšího z nich. Článek [2] uvádí další možnosti pro dosažení akcelerace výpočtů u evolučních algoritmů na heterogenních systémech.

Kapitola 9

Závěr

Systémy na čipu a zejména pak multiprocesorové systémy na čipu (MPSoC [10], [21]) se v současnosti velmi rozšiřují. Tento trend je podporován velkými výrobci hardware, jako jsou Tilera nebo Intel. Důkazem toho je například generace procesorů Intel Sandy Bridge [25], kde jsou jednotlivá jádra procesoru spojena do kruhové topologie pomocí 128 bitové sběrnice. Lze tudíž předpokládat, že spolu s rozvojem MPSoC bude probíhat rozvoj propojovacích sítí a spolu s nimi bude růst význam algoritmů pro hledání bezkolizních komunikačnách plánů.

V rámci této práce byl proveden podrobný rozbor a časová analýza existující aplikace implementující evoluční algoritmus pro hledání bezkolizních plánů kolektivních komunikací na propojovacích sítích [11]. Bylo potvrzeno, že hodnotící funkce fitness dominuje celému výpočtu z hlediska časové náročnosti. Navržené řešení (viz kapitola 5) paralelního zpracování této části algoritmu bylo následně implementováno. Po zavedení důležitých optimalizací (viz sekce 6.2), zaměřených zejména na práci s pamětí, bylo provedeno testování a porovnání s původní implementací pro CPU (viz kapitola 7).

Z výsledků testování na různých topologiích uvedených v 7.2 vyplývá, že využití této aplikace, akcelerované pomocí paralelního zpracování na GPU, má smysl pokud jsou řešeny rozsáhlejší topologie, tedy topologie s alespoň 30 procesory. Dalé je potřeba aplikaci využívat s moderními grafickými kartami. Výkon aplikace samozřejmě závisí na výpočetním výkonu daného GPU, tedy zejména na počtu výpočetních jader a dále na kapacitě pamětí a rychlosti přístupů do jednotlivých typů pamětí. Například testovaná grafická karta nVidia Quadro FX 770M se ukázala jako prakticky nepoužitelná z důvodu nízkého výpočetního výkonu a také z důvodu nedostatečného chlazení, což je problém u většiny grafických karet v přenosných počítačích. Maximální zrychlení na jedné grafické kartě GeForce GTX 580 proti procesoru Intel Core i7 920 bylo naměřeno na topologii mřížka 7x7. Toto zrychlení dosáhlo hodnoty 4.81.

Vytvořené řešení dosáhlo významného zrychlení. Jelikož lze ale předpokládat stálé zvyšování počtu procesorů v propojovacích sítích bude toto zrychlení nedostatečné. Z tohoto důvodu je v části 9.2 popsán možný směr dalšího vývoje této aplikace.

Vzhledem ke způsobu porovnávání popsaném v části 6.2.1, kdy každé vlákno porovnává jedinou dvojici cest, jsou v topologiích s desítkami procesorů vytvářeny desítky až stovky tisíc vláken pro každého ohodnocovaného jedince v populaci. Díky tomu dochází k dostatečnému vytížení GPU i v populacích čítajících jediného jedince. Jelikož je současným trendem v evolučních algoritmech snižování velikosti populace ([28], [13]), je toto zjištění důležité pro možné budoucí využití tohoto způsobu řešení.

9.1 Shrnutí výsledků

V následujících bodech jsou stručně shrnuty hlavní části práce a výsledky, kterých bylo dosaženo. Popsané hodnoty zrychlení a další výsledky testování se vztahují ke grafické kartě nVidia GeForce GTX 580 a k procesoru Intel Core i7 920 při čtyřvláknovém zpracování.

1. Analýza existující aplikace – funkce `computeFitnessValue()` identifikována jako časově nejnáročnější část. Podle grafu 5.1 zde program stráví 97.21 % doby výpočtu. Ohodnocení populace o velikosti 200 jedinců v topologii mřížka 7x7 trvá na CPU přibližně 4.5 vteřiny.

2. Návrh paralelního zpracování evolučního algoritmu (5.1) – na základě výsledků analýzy navržena pro paralelní zpracování pouze funkce `computeFitnessValue()`. Poměrně dlouhý čas výpočtu pro ohodnocení jedné generace překryl režii potřebnou pro spuštění výpočtu na GPU. Odhad provedený v části 5.3 určil teoreticky dosažitelné zrychlení mezi použitým GPU a CPU na 8.

3. Implementace paralelní verze pro GPU v OpenCL (6) – vytvořeny moduly v C++ realizující rozhraní mezi existující aplikací a částí zpracovávanou na GPU. Zajišťují inicializaci GPU a přesun vstupních a výstupních dat. Kernel pro výpočet na GPU implementující funkci fitness je vytvotřen v jazyce OpenCL C.

4. Byla navržena a implementována řada optimalizací (6.2) – paralelní redukce, využití lokalní paměti, dopředná alokace paměti, zvýšení počtu vláken (vytížení GPU), zavedení třídimenzionálních bloků (paralelní zpracování na úrovni populace), minimalizace přístupů do globální paměti (dopředné načítání cest do privátní paměti), algoritmus pro přidělení jedinečné dvojice cest každému vláknu

5. Testování aplikace a určení zrychlení na GPU proti CPU

   - Provedeno komplexní testování na mnoha topologiích s různými parametry. (7.2)
   - Prezentace důležitých výsledků v přehledných grafech a tabulkách.
   - Optimální velikost bloku testováním určena na (1024,1,1). (7.1)
   - Aplikace dosáhla na větších topologiích téměř pětinásobného zrychlení.

6. Rozšíření na více platforem

   - Aplikace pracuje se všemi zařízeními podporovanými OpenCL.
   - Podpora distribuce výpočtu na všechna zařízení v systému.
   - Testování multi GPU režimu – zrychlení 31x na 7x GPU proti CPU. (8.2)

9.2 Budoucí vývoj

Velikost propojovacích sítí se bude s vývojem procesorů a systémů na čipu stále zvyšovat. Z tohoto důvodu bude nutné stejným způsobem zvyšovat rychlost hledání bezkolizních komunikačních plánů. Dosažené zrychlení poměrně výrazně zkrátí dobu výpočtu a umožní například najít bezkolizní plán pro topologii 5D kostka za jeden den a devět hodin místo původních čtyř dnů a pěti hodin (viz tabulka 7.1). Implementované rozšíření distribuce výpočtu mezi více GPU v systému naznačuje možný směr dalšího vývoje. Kromě multi

GPU zpracování může být výpočet distribuován na heterogenní zařízení v systému, popřípadě distribuován v rámci svazku výpočetních stanic propojených počítačovou sítí. V tomto případě bude nutné uvažovat vysokou režii, která vznikne zpomalením komunikace po počítačové síti. Dále bude nutné efektivně řešit problém rozdělení zátěže mezi různě výkonná vypočetní zařízení.

Na základě provedené časové analýzy byla pro paralelní zpracování na GPU převedena pouze funkce `computeFitnessValue()`, jenž představovala přes 97 % doby výpočtu. Dosažením téměř pětinásobného zrychlení (resp. až 31násobného zrychlení při použití sedmi GPU) se však tento poměr mění a doba strávená v sekvenčně prováděné části aplikace výrazně roste. Budoucím rozšířením stávajícího řešní by tedy mohla být implementace celého evolučního algoritmu na GPU. Bylo by nutné vytvořit podrobný návrh a odhad možného dosažitelného zrychlení, na jehož základě by bylo možné rozhodnout, jestli se převod celého evolučního algoritmu na GPU vyplatí.

Práce se podrobně zabývá akcelerací kolektivních komunikací typu *all to all scatter*. Mezi další často používané typy kolektivních komunikací v paralelních aplikacích patří *one to all scatter*, *one to all broadcast* a *all to all broadcast*, které jsou také řešeny v [11]. Jelikož se ve všech případech jedná o hledání konfliktů mezi naplánovanými cestami, může být řešení implementované v této práci, poměrně snadno rozšířeno i na tyto další typy kolektivních komunikací.

Literatura

[1] *CSTST '08: Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-046-3.

[2] Alba, E.; Nebro, A. J.; Troya, J. M.: Heterogeneous computing and parallel genetic algorithms. *J. Parallel Distrib. Comput.*, ročník 62, č. 9, Září 2002: s. 1362–1385, ISSN 0743-7315, doi:10.1006/jpdc.2002.1851.
URL http://dx.doi.org/10.1006/jpdc.2002.1851

[3] Beyer, H.-G.; Schwefel, H.-P.: Evolution strategies – A comprehensive introduction. ročník 1, 2002: s. 3–52, ISSN 1567-7818.
URL http://dl.acm.org/citation.cfm?id=584639.584641

[4] de Castro, L. N.: *Fundamentals of Natural Computing (Chapman & Hall/Crc Computer and Information Sciences)*. Chapman & Hall/CRC, 2006, ISBN 1584886439.

[5] Dally, W. J.; Towles, B.: Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, ACM, 2001, ISBN 1-58113-297-2, s. 684–689.
URL http://doi.acm.org/10.1145/378239.379048

[6] Farber, R.: OpenCL – Memory Spaces. 2010.
URL
http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces

[7] Hancock, P.: An empirical comparison of selection methods in evolutionary algorithms. In *Evolutionary Computing*, *Lecture Notes in Computer Science*, ročník 865, Springer Berlin / Heidelberg, 1994, ISBN 978-3-540-58483-4, s. 80–94.
URL http://dx.doi.org/10.1007/3-540-58483-8_7

[8] Holland, J. H.: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. A Bradford Book, 1975, ISBN 978-0262581110.

[9] Hynek, J.: *Genetické algoritmy a genetické programování*. Grada Publishing, 2008, ISBN 978-80-247-2695-3.

[10] Hübner, M.; Becker, J.: *Multiprocessor System-on-Chip*. Springer-Verlag, 2011, ISBN 978-1-4419-6459-5.

[11] Jaroš, J.: *Evolutionary Design of Collective Communications on Wormhole Networks*. Dizertační práce, FIT VUT, Brno, 2010.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9260

[12] Jaroš, J.; Treeby, E. B.; Rendell, P. A.: Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations. In *Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012)*, 2012, s. 43–52.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9881

[13] Jin, Y.; Olhofer, M.; Sendhoff, B.: On evolutionary optimization of large problems using small populations. In *Proceedings of the First international conference on Advances in Natural Computation - Volume Part II*, ICNC'05, Berlin, Heidelberg: Springer-Verlag, 2005, ISBN 3-540-28325-0, 978-3-540-28325-6, s. 1145–1154.
URL http://dx.doi.org/10.1007/11539117_154

[14] Khronos OpenCL Working Group: *The OpenCL Specification*. 2011, verze 1.1.44.
URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[15] Kirk, D. B.; Hwu, W.-m. W.: *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010, ISBN 978-0-12-381472-2.

[16] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford Book, 1992, ISBN 978-0262111706.

[17] Kvasnička, V.; Pospíchal, J.; Peter, T.: *Evoluční algoritmy*. Bratislava: STU, 2000, ISBN 80-227-1377-5.

[18] Langdon, W. H.: *Genetic Programming and Data Structures*. Kluwer Academic Publishers, 1998, ISBN 978-0-7923-8135-8.

[19] Leung, J.: *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman & Hall/CRC, 2004, ISBN 9781584883975.

[20] nVidia Corporation: *CUDA C Best Practices Guide*. 2012, verze 4.1.
URL developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

[21] Popovici, K.; Rousseau, F.; Jerraya, A.: *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer-Verlag, 2010, ISBN 978-1-4419-5566-1.

[22] Schwarz, J.; Sekanina, L.: *Aplikované evoluční algoritmy*. Brno: FIT VUT, 2008.

[23] Sekanina, L.; aj.: *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Praha: Academia, 2009, ISBN 978-80-200-1729-1.

[24] Urminský, A.: *Použití evolučního algoritmu ve hře šachy*. Diplomová práce, FIT VUT, Brno, 2007.

[25] Varis, N.; Manner, J.: In the network: sandy bridge versus nehalem. *SIGMETRICS Perform. Eval. Rev.*, ročník 39, č. 2, Září 2011: s. 53–55, ISSN 0163-5999.
URL http://doi.acm.org/10.1145/2034832.2034846

[26] Wilson, G. V.: A Glossary of Parallel Computing Terminology. *IEEE Parallel Distrib. Technol.*, ročník 1, February 1993: s. 52–67, ISSN 1063-6552.
URL http://dl.acm.org/citation.cfm?id=613767.613799

[27] Wittenbrink, C. M.; Kilgariff, E.; Prabhu, A.: Fermi GF100 GPU Architecture. *IEEE Micro*, ročník 31, č. 2, Březen 2011: s. 50–59, ISSN 0272-1732.
URL http://dx.doi.org/10.1109/MM.2011.24

[28] Yi, H.; Sam, K.; Qingsheng, R.: Over-Selection: An attempt to boost EDA under small population size. In *IEEE Congress on Evolutionary Computation*, CEC 2007, IEEE Computer Society Press, 2007, ISBN 978-1-4244-1340-9, s. 1075 – 1082.

[29] Zelinka, I.; aj.: *Evoluční výpočetní techniky – principy a aplikace*. Praha: BEN, 2009, ISBN 978-80-7300-218-3.

[30] Zhang, E. Z.; Jiang, Y.; Guo, Z.; aj.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0018-6, s. 115–126.
URL http://doi.acm.org/10.1145/1810085.1810104

Příloha A

Obsah přiloženého CD

`/bin/scatter-cpu/`	zkompilovaná verze aplikace pro CPU
`/scatter-gpu/`	zkompilovaná verze aplikace pro GPU
`/doc/tex/`	zdrojové soubory textu diplomové práce
`/manual.txt`	popis zprovoznění a ovládání aplikace
`/projekt.pdf`	text diplomové práce
`/src/scatter-cpu/`	zdrojové soubory verze pro CPU
`/scatter-gpu/`	zdrojové soubory verze pro GPU
`/tests/test-cpu`	ukázka původní aplikace - ohodnocování na CPU
`/test-gpu`	ukázka upravené aplikace - ohodnocování na GPU
`/test-all`	ukázka distribuce výpočtu mezi všechna zařízení v systému
`/topologies/`	soubory s testovacími topologiemi

Příloha B

Zprovoznění a ovládání aplikace

Aplikace je vytvořena v jazyce C++ v prostředí operačního systému Linux a otestována v distribucích Arch (64 bit), CentOS a Ubuntu. Pro zprovoznění aplikace je nutné mít nainstalovaný ovladač grafické karty, umožňující provádění GPGPU výpočtů, včetně příslušného kompilátoru. Aplikace byla testována s ovladači nVidia 285.05.33, 295.20 a 295.40 a s CUDA toolkitem obsahujícím OpenCL verze 1.1. Z důvodu vícevláknového zpracování na CPU je vyžadována knihovna OpenMP.

Zdrojové kódy jsou rozděleny na dvě části, verzi pro CPU a verzi pro GPU resp. pro všechna zařízení podporovaná OpenCL. Zdrojové texty jsou vybaveny Makefilem, pro jejich překlad tedy stačí zadat příkaz `make`. Jedná se o konzolovou aplikaci, která očekává devět parametrů popsaných v tabulce B.1. V případě spuštění aplikace s nesprávným počtem parametrů je na standardní výstup vypsána nápověda a aplikace je ukončena. Výkon aplikace je možné vyladit pro konkrétního zařízení pomocí konstant popsaných v tabulce B.2.

Tabulka B.1: Popis očekávaných parametrů

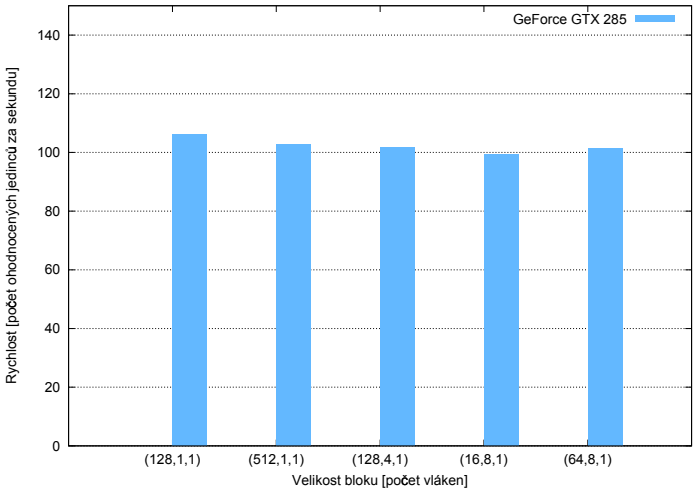
Pořadí	Popis parametru	Ukázková hodnota
1	*soubor s popisem topologie*	`../../topologies/Mesh/mesh-5x5.txt`
2	*počet komunikačních kroků*	32
3	*velikost populace*	100
4	*maximální počet generací*	50 000
5	*míra mutace*	0.8
6	*maximální prodloužení cesty*	0
7	*počet vláken na CPU*	4

Tabulka B.2: Konstanty ovliviňující chování aplikace

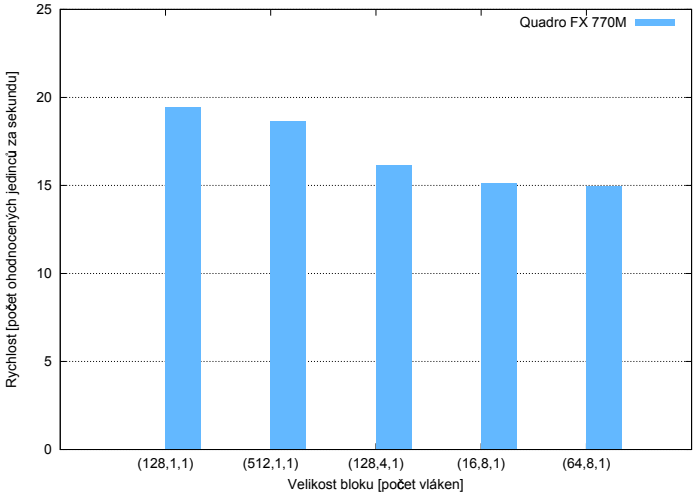
Název konstanty	Výchozí hodnota	Popis konstanty
`LOCAL_WORK_SIZE_X`	1024	počet dvojic cest na jednom bloku
`LOCAL_WORK_SIZE_Y`	1	počet časových kroků na jednom bloku
`CL_FILE_PATH`	`"./fitness.cl"`	relativní cesta ke zdrojovému kódu kernelu
`MAX_CHROMS_PER_CYCLE`	1000	max. počet jedinců ohodnocovaných současně
`GET_ALL_DEVS`	`false`	při `true` použita všechna zařízení

Příloha C

Testování velikosti bloku



Obrázek C.1: Rychlost ohodnocování v závislosti na velikosti bloku pro GeForce GTX 285.



Obrázek C.2: Rychlost ohodnocování v závislosti na velikosti bloku pro Quadro FX 770M.